doubt

EXTRA THINGS OF PYTHON TO LEARN

1) Course curriculum

1) PYTHON : MAKE HTTP REQUEST .
   - TTP Introduction
   - HTTP verbs and API
   - Writing your First Python Request
   - Requesting Json with Python
   - Sending Request with Params
   - API Projects

2) PYTHON : DECORATORS .
   - Higher Order Functions
   - Introduction to Decorators
   - Decorators with Different Signature
   - Using Wraps to preserver Metadata
   - Building a Speed Test Decorator

3) PYTHON : TESTING.
   - Section Introduction
   - Why Test
   - Assertions
   - Introduction to Doctest
   - Introduction to Unit Test
   - Hands on work on Testing

4) PYTHON: FILE INPUT AND OUTPUT

- Section Introduction
- Reading the Text File: Open, read and Write
- Reading the File: Seek and Cursor
- The with Statement
- Writing to Text File
- Working on Csv and Pickling
- Reading CSV File
- Writing CSV File

5) PYTHON : WEB SCRAPING WITH BEAUTIFUL SOUP .
   - Introduction to Web Scraping
   - Is Scraping ok?
   - Selecting with Beautiful Soup
   - Navigate with Beautiful Soup
   - Web Scraping Projects

6) PYTHON: REGULAR EXPRESSION
   - Introduction to Regular Expression
   - Writing basic Regix

- Regix basic Quantifiers

7) PYTHON: PYTHON + SQL.
    - Introduction to the section
    - Installing Mysql
    - Why Mysql
    - Sql Basic: DDl , DML
    - Inserting with Python
    - Fetching the Data in Python from MySQL

8) PYTHON: Libraries .
    - Numpy
    - Lambda
    - Etc for data analysis

coding game

1) Codedex =python
2) Codin game = basic and advance java
3) Elevator saga = java

Import Random
Or
From random import randint – only for (random.randint)

Case 1 :
Import Random

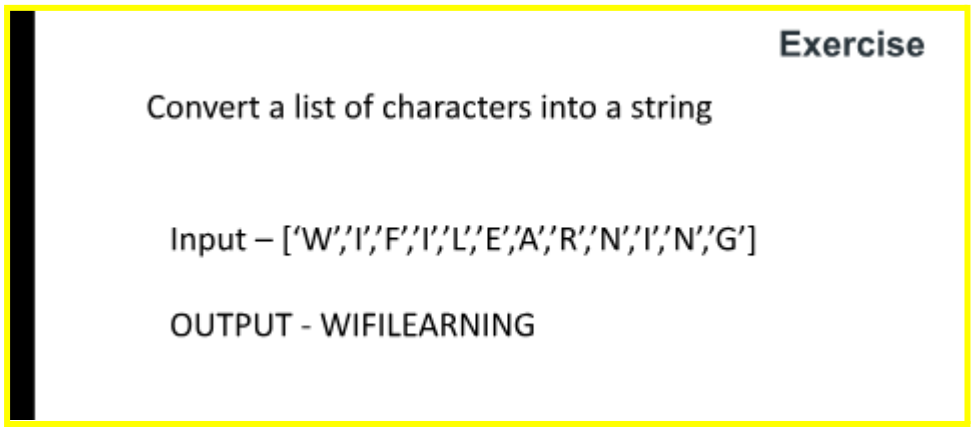
Random.randint
Ranodm.choice(list)

Or

From random import randint – only for (random.randint)
randint
Ranodm.choice(list) — it can’t be work as you only import randint

Py_questions

Q.1



Sol 1 -

```
li = ['w', 'i', 'f', 'i', 'l', 'e', 'a', 'r', 'n', 'i', 'n', 'g']
result = "".join(li)  #here “” this showing that there should be gap of (like “-”, “ “ - means dash)
print(result)
```

_____

Sol 2 -

```
li = ['w', 'i', 'f', 'i', 'l', 'e', 'a', 'r', 'n', 'i', 'n', 'g']
a = str()
for i in li:!
    a += i # += means for add, its means a = a + i.
print(a)
print(a[2])
```

****

Q. 2

1. Reversing a Tuple

2 .Count occurrences of an element in a Tuple

Tuple: (10, 8, 5, 2, 10, 15, 10, 8, 5, 8, 8, 2)
Input - 10
Out put - 3

Sol 1 - print(tuple1(: : -1)

Sol 2 -

```
Tuple = (10, 8, 5, 2, 10 ,14, 10)
print(Tuple)
a = int(input("Enter the number : "))
if a in Tuple:
    z = Tuple.count(a)
    print(z)
else:
    print("The number is not in Tuple")
```

Another method withoud use (count method)

```
Tuple = (10, 8, 5, 2, 10 ,14, 10)
lis = []
a = int(input("Enter the number : "))
if a in Tuple:
    for i in Tuple:
        if (a == i):
            lis.append(i)
    print(len(lis))
else:
    print("The number is not in Tuple")
```

****

Q 3 Interview que - How to reverse a tuple
Ans - print(tuple(: : -1))

****

Q 4 How to Remove () blank tuple from list

```
Tuple = [(), ('ram','15','8'), (), ('laxman', 'sita'), ('krishna', 'akbar', '45'), ("",),()]
Tuple1 = ()
for i in Tuple:
    if i == Tuple1:
        Tuple.remove(i)
print(Tuple)
```

****

Q.5

Exercise 2 - Python - Append Dictionary Keys and Values (In order) in dictionary
Input : test_dict = {"Gfg" : 1, "is" : 2, "Best" : 3}
Output : ['Gfg', 'is', 'Best', 1, 2, 3]
Explanation : All the keys before all the values in list.

Sol :

```
test = {'gfg':1,"is":2,"best":3}
a = test['gfg']
b = test['is']
c = test["best"]
z = [a,b,c]
```

```
y = list(test)
print(y+z)
```

Output : ['gfg', 'is', 'best', 1, 2, 3]

Or _____

```
test = {'gfg':1,'is':2,"best":3}
res = list(test.keys()) + list(test.values())
print(str(res))
print(type(res))
print(type(test))
```

Output =
['gfg', 'is', 'best', 1, 2, 3]
<class 'list'>
<class 'dict'>

Q.6

Exercise 3 – Python program to update a dictionary with the values from a dictionary list

Input : test_dict = {"Gfg" : 2, "is" : 1, "Best" : 3}, dict_list = [{'for' : 3, 'all' : 7}, {'and' : 1, 'CS' : 9}]
Output : {'Gfg': 2, 'is': 1, 'Best': 3, 'for': 3, 'all': 7, 'and': 1, 'CS': 9}
Explanation : All dictionary keys updated in single dictionary.

```
test = {'gfg':1,'is':2,"best":3}
test_list = [{"for":3,"all":7},{"and":1,"Cs":9}]
for dicts in test_list:
    test.update(dicts)
print(test)
print(len(test_list))
```

Output =

{'gfg': 1, 'is': 2, 'best': 3, 'for': 3, 'all': 7, 'and': 1, 'Cs': 9}
2

Q.7 . Print Even Numbers from 2 to 20 .

```
i = 1
while(i<=20):
    if (i%2 == 0):
        print(i)
    i += 1
```

Or

```
i = 2
while(i<=20):
    print(i)
    i += 2
```

Q.8 . Find the Factorial of a Number by while loop

```
inp = int(input("Enter the digit : "))
i = 1
mul = 1
while(i<=inp):
    mul = mul*i
    i += 1
print("factorial of",inp,"! is",mul)
```

Q.9 Count Digits in a Number

```
num = int(input("Enter the digit : "))
count = 0
while(num > 0):
    num = num//10
    count += 1
    print(count)
    print(num)
print(count)
```

****

```
num = int(input("Enter the digit : "))
numstr =  str(num)
li = list(numstr)
print("Total digit in ",num,"is ",len(li))
```

******

Q.10 Count of particular Digits in a Number

```
inp = int(input("Enter Number : "))
no = int(input("Enter digit which you count : "))
num = str(no)
```

```
li = str(inp)
fi = list(li)
print(fi)
print(fi.count(num))
```

Q. 11 Count all unique Digits in a Number

```
num = int(input("Enter the digit : "))
numstr =  str(num)
li = set(numstr)
print("Total digit in ",num,"is ",len(li))
```

Q. 12 Find the Sum of Digits of a Number

Mine projects

1) . Enter table of any number .

```
k = int(input("Here multiplication table, Enter number : "))
for i in range(1,11):
    print(k,"x",i,"=",k*i)
```

2) . Prime no list and non prime no list .

```
n = int(input("Enter no : "))

if (n>1):
    for i in range(2,n):
        if(n%i == 0):
            print(f"{n} is not prime no")
            break
    else:
        print(f"{n} is prime no")
else:
    print(f"{n} is not prime no")
```

- not working its static type

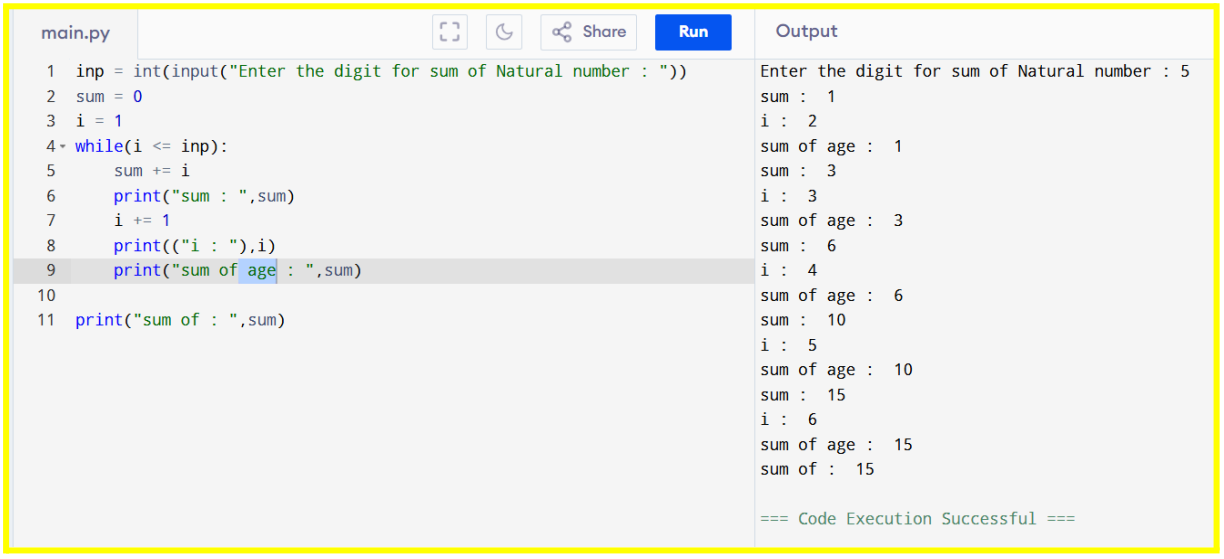
```
lis = [ ]
lis1 =[ ]
for i in range(1,11):
    if (i%2 == 0 or i%3 == 0 or i%5 == 0 or i%7 == 0):
        lis.append(i)
lis.remove(2)
lis.remove(3)
for k in range(2,11):
    if (k not in lis):
        lis1.append(k)
print("The non prime no List is :",lis)
print("The prime no List is :",lis1)
```

3) sum of all whole number till an input number

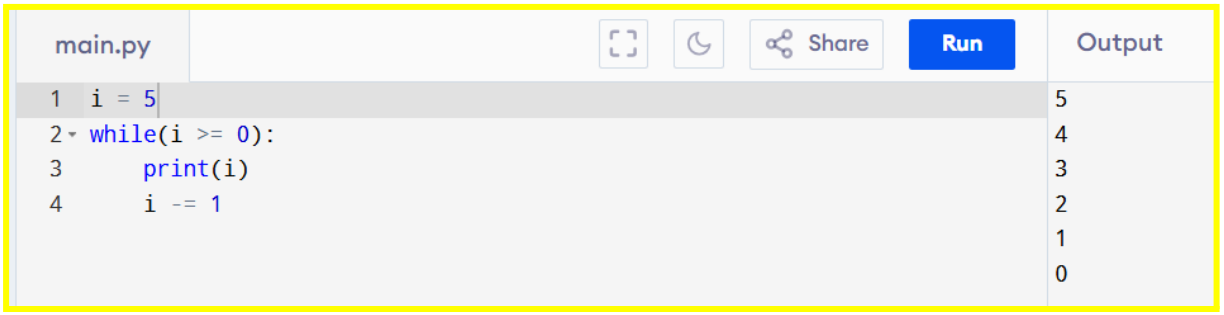
```
inp = int(input("Enter the Number at which you want to add : "))
li = [ ]
for k in range(inp+1):
    li.append(k)
#print(li)
a = int()
for i in li:
    a += i
print(a)
```

4) Sum of First N Natural Numbers by While loop.

```
inp = int(input("Enter the digit for sum of Natural number : "))
sum = 0
i = 1
while(i <= inp):
    sum += i
    i += 1
print("sum of : ",sum)
```


main.py
Share
Run
Output
1 inp = int(input("Enter the digit for sum of Natural number : "))
2 sum = 0
3 i = 1
4 while(i <= inp):
5 sum += i
6 print("sum : ",sum)
7 i += 1
8 print(("i : "),i)
9 print("sum of age : ",sum)
10
11 print("sum of : ",sum)
Enter the digit for sum of Natural number : 5
sum : 1
i : 2
sum of age : 1
sum : 3
i : 3
sum of age : 3
sum : 6
i : 4
sum of age : 6
sum : 10
i : 5
sum of age : 10
sum : 15
i : 6
sum of age : 15
sum of : 15
=== Code Execution Successful ===

5. **Reverse a Number by while loop**



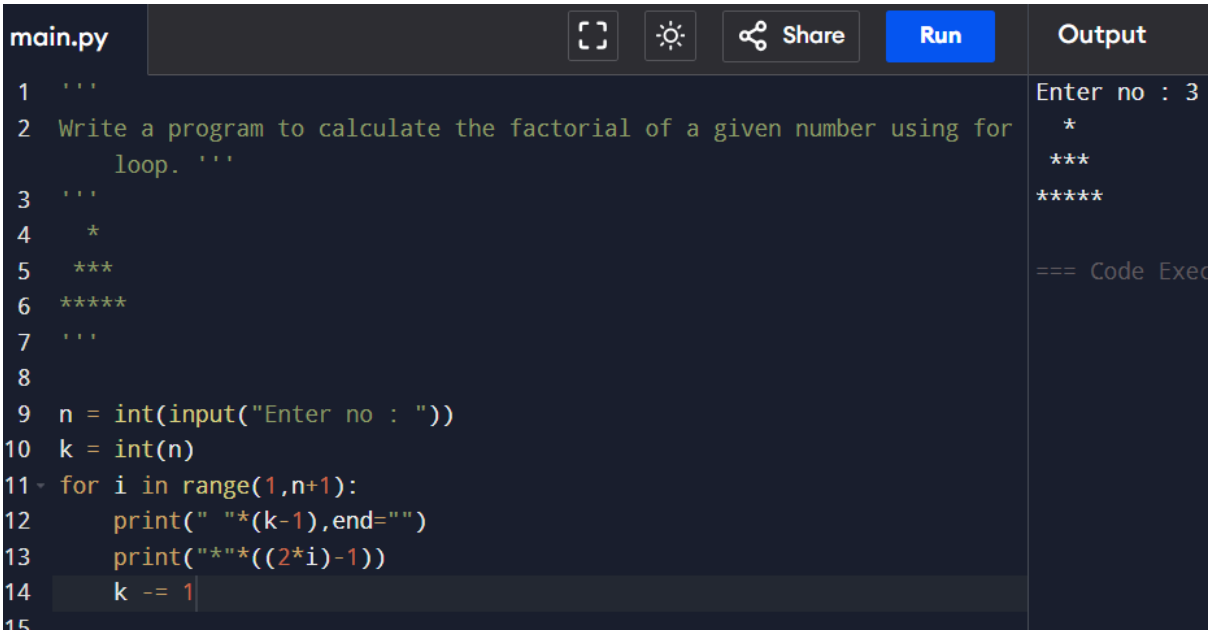
6. Guess the Secret Number Game

```
print(" Hi Buddy!!!, Let's Play a Guessing Game \n Here the following is some specific Rules - \n - just you have to guess a number Unitl your number is match as guessing number, you will win the game")
print("\n")

inp = 0
while(inp != 9):
    inp = int(input("Enter Your Guessing Number : "))
    if(inp != 9):
        print("Wrong number, please enter again","\n")
    else:
        print("\nYou won,\nGame over")
```

6. **Multiple of any number till 10 digits**

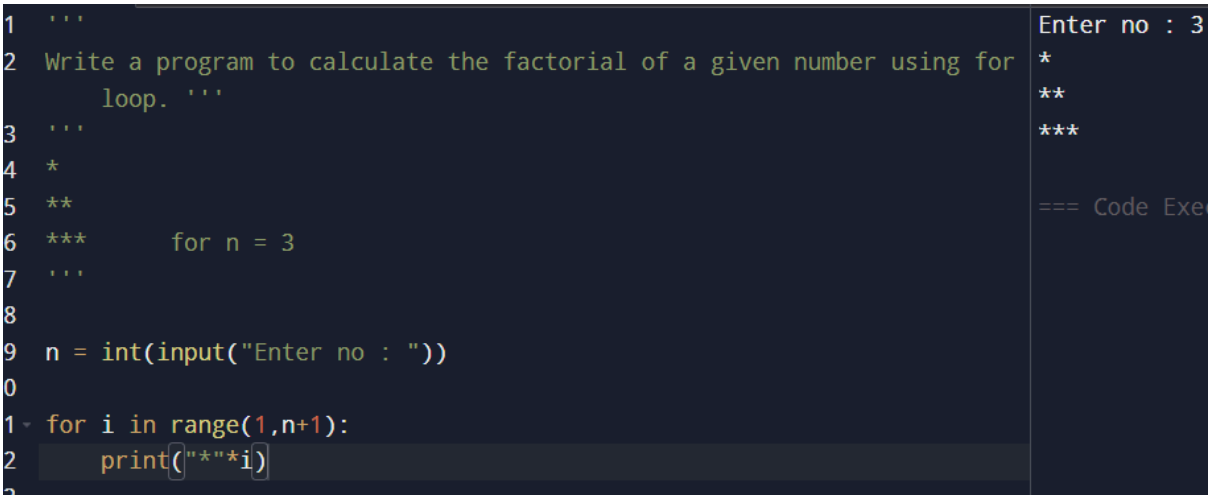
```
num = int(input("Enter the number : "))

num1 = 0
while (num1 <= 9*num):

    num1 = num1 + num
    print(num1)
```

7. Write a program to print the following star pattern.




```
n = int(input("Enter no : "))
k = int(n)
for i in range(1,n+1):
    print(" "*(k-1),end="")
    print("*"*((2*i)-1))
    k -= 1
```

8. Write a program to print the following star pattern.




```
n = int(input("Enter no : "))

for i in range(1,n+1):
    print("*"*i)
```

9. Write a program to print the following star pattern.

```
n = int(input("Enter no : "))

for i in range(1,n+1):
    if (i == 1 or i == n):
        print("*"*n)
    else:
        print("*",end="")
        print(" "*(n-2),end="")
        print("*")
```

10. Write a pfadf

cl - 1

❖ Python Comments

1. Single-Line Comments:

Eg -

```
#This is a 'Single-Line Comment'
print("This is a print statement.")
```

2. Multi-Line Comments:

Eg 1-

```
#It will execute a block of code if a specified condition is true.
#If the codition is false than it will execute another block of code.
```

Eg 2-

```
"""This is an if-else statement.
It will execute a block of code if a specified condition is true.
If the condition is false then it will execute another block of code."""
```

❖ Variables - Data Types

- Basic Data Types

1) type str (string) - eg. name = "Abhishek"
2) type int (integer) - eg. age = 20
3) type bool (boolean) - eg. passed = True
4) float - e.g., 3.14, 2.718

- Collection Data Types:

1) List [] : Ordered, mutable collection of elements (e.g., [1, 2, "Hello"])
2) Tuple () : Ordered, immutable collection of elements (e.g., (1, 2, "Hello"))
3) Dict {} : Key-value pairs (e.g., {"name": "Alice", "age": 30})
4) Set {} : Unordered collection of unique elements (e.g., {1, 2, 3})
5) Frozenset {} : Immutable set (e.g., frozenset({1, 2, 3}))
    - Only tuple, and frozen set immutable(can't change) .

Two types of variables scope

1) Local Variable

2) Global Variable

```
icecream = "Vanilla"    #global variable
def foods():
  vegetable = "Potato"    #local variable
  fruit = "Lichi"         #local variable
  print(vegetable + " is a local variable value.")
  print(icecream + " is a global variable value.")
  print(fruit + " is a local variable value.")
```

For find data types = print(type(state))

❖ Line break

1) use = \n

- ❖ list.remove() - it removes that number, not that place
- ❖ list.pop() - it remove the nth positions of list

Homogeneous:

- A data structure is considered homogeneous if all its elements are of the same data type.
- Example: A list containing only integers `[1, 2, 3, 4]` is homogeneous.

Heterogeneous:

- A data structure is considered heterogeneous if its elements can be of different data types.
- Example: A list containing integers, strings, and floats `[1, "hello", 3.14]` is heterogeneous.

Feature	for Loop	while Loop
Use Case	Used when the number of iterations is known.	Used when the number of iterations is unknown and depends on a condition.
Syntax	Uses an iterator or range() function.	Uses a condition that must be True to continue execution.
Loop Control	Stops when the iteration reaches its limit.	Stops when the condition becomes False.
Best For	Iterating over sequences (lists, tuples, dictionaries, strings, ranges, etc.).	Running a block of code repeatedly until a condition is met.

Why we use generator

get_ipythonget_ipython().set_next_input('Summary: Why Use Generators');get_ipython().run_line_magic('pinfo', 'Generators')

✅ Memory Efficient → They generate values on the fly instead of storing them in memory.
✅ Lazy Evaluation → They compute the next value only when requested, making them great for large datasets.
✅ Infinite Sequences → Can generate infinite values without memory overflow.
✅ Better Performance → Faster than returning large lists in many cases.

what we have to write in new terminal for run the python code in vs code

1) we will enter there location in quotetion, like - cd '_____'
2) python filename.py —> ✅ Example: python .\hello.py

What

Python built in fn

Python Built-in Functions

Python provides numerous built-in functions that perform common tasks, making programming more efficient and easier. Here is a categorized list of all Python built-in functions:l

---

1. Input/Output Functions

- `print()`: Outputs data to the console.
- `input()`: Reads user input as a string.
- `open()`: Opens a file and returns a file object.

Example:

```
name = input("Enter your name: ")
print(f"Hello, {name}!")
```

---

2. Type Conversion Functions

Convert data from one type to another.

- `int()`: Converts a value to an integer.
- `float()`: Converts a value to a float.
- `str()`: Converts a value to a string.
- `list()`: Converts an iterable to a list.
- `tuple()`: Converts an iterable to a tuple.
- `set()`: Converts an iterable to a set.
- `dict()`: Converts a sequence of key-value pairs into a dictionary.
- `bool()`: Converts a value to a boolean.

Example:

```
num = "10"
print(int(num) + 5)  # Output: 15
```

---

3. Mathematical Functions

Perform mathematical operations.

- `abs()`: Returns the absolute value.
- `round()`: Rounds a number to a specified number of decimals.
- `pow()`: Returns the power of a number.
- `min()`: Returns the smallest element.
- `max()`: Returns the largest element.
- `sum()`: Returns the sum of elements in an iterable.

Example:

```
numbers = [1, 2, 3, 4]
print(sum(numbers))  # Output: 10
```

---

4. Sequence and Iterable Functions

Operate on sequences and iterables.

- `len()`: Returns the length of a sequence.
- `sorted()`: Returns a sorted version of an iterable.
- `reversed()`: Returns a reversed iterator.
- `enumerate()`: Returns an iterator of index-value pairs.
- `zip()`: Combines multiple iterables element-wise.
- `range()`: Generates a sequence of numbers.
- `filter()`: Filters elements of an iterable based on a condition.
- `map()`: Applies a function to each element of an iterable.
- `all()`: Returns `True` if all elements are true.
- `any()`: Returns `True` if any element is true.

Example:

```
names = ["Alice", "Bob", "Charlie"]
for i, name in enumerate(names):
    print(i, name)
```

---

5. String Functions

- `chr()`: Converts an integer to its Unicode character.
- `ord()`: Converts a character to its Unicode code.

Example:

```
print(chr(65))  # Output: A
print(ord('A'))  # Output: 65
```

---

6. Object and Attribute Functions

- `type()`: Returns the type of an object.
- `id()`: Returns the memory address of an object.
- `isinstance()`: Checks if an object is an instance of a class.
- `hasattr()`: Checks if an object has an attribute.
- `getattr()`: Retrieves the value of an attribute.
- `setattr()`: Sets an attribute's value.
- `delattr()`: Deletes an attribute.
- `dir()`: Returns the attributes of an object.
- `callable()`: Checks if an object is callable.

Example:

```
class MyClass:
    x = 10

obj = MyClass()
print(hasattr(obj, 'x'))  # Output: True
```

---

7. Advanced Functions

- `eval()`: Parses and evaluates a string as Python code.
- `exec()`: Executes a block of Python code.
- `compile()`: Compiles source code into an executable object.

Example:

```
code = "print('Hello, World!')"
eval(code)  # Output: Hello, World!
```

---

8. Miscellaneous Functions

- `help()`: Provides the documentation of an object.
- `globals()`: Returns a dictionary of the current global symbol table.
- `locals()`: Returns a dictionary of the current local symbol table.
- `input()`: Takes input from the user.
- `len()`: Returns the length of an object.

Example:

```
help(print)
```

---

9. File Handling Functions

- `open()`: Opens a file.
- `close()`: Closes an open file.

Example:

```
with open('file.txt', 'w') as file:
    file.write('Hello, World!')
```

This is a comprehensive overview of Python's built-in functions. For more detailed usage, refer to the Python documentation.

Python List Methods

Python List Methods

Find list elements positions.

List = ["ram","you","we"]
print(List[0]) = ram

Lists in Python have a variety of built-in methods for common operations:

Adding Elements

- `append(x)`: Adds an element to the end of the list.
- `extend(iterable)`: Adds all elements of an iterable to the list.
- `insert(i, x)`: Inserts an element at a specified position.

Removing Elements

- `remove(x)`: Removes the first occurrence of the element.
- `pop([i])`: Removes and returns the element at the specified position (default is the last element).
- `clear()`: Removes all elements from the list.

Querying Elements

- `index(x[, start[, end]])`: Returns the index of the first occurrence of an element.
- `count(x)`: Returns the number of occurrences of an element.

Sorting and Reversing

- `sort(key=None, reverse=False)`: Sorts the list in place.
- `reverse()`: Reverses the list in place.

Copying

- `copy()`: Returns a shallow copy of the list.

Example:

```
my_list = [3, 1, 4, 1, 5]
my_list.append(9)
print(my_list)  # Output: [3, 1, 4, 1, 5, 9]
```

```
my_list.sort()
print(my_list)  # Output: [1, 1, 3, 4, 5, 9]
```

Comprehensive List of Python List Methods

Method	Description
`append(x)`	Adds an item to the end of the list.
`extend(iter)`	Extends the list by appending elements from the iterable.
`insert(i, x)`	Inserts an item at a given position.
`remove(x)`	Removes the first occurrence of an item.
`pop([i])`	Removes and returns the item at the given position (last item by default).
`clear()`	Removes all items from the list.
`index(x)`	Returns the index of the first occurrence of an item.
`count(x)`	Returns the number of occurrences of an item.
`sort()`	Sorts the list in ascending order (can be customized with key/reverse).
`reverse()`	Reverses the list in place.
`copy()`	Returns a shallow copy of the list.

This is a comprehensive overview of Python's built-in functions and list methods. For more detailed usage, refer to the Python documentation.

random imp code

1.) Code -

```
tu = tuple('geek')
print(tu)
print(tu[1])
```

Output -

```
('g', 'e', 'e', 'k')
e
```

- Its also give same response like list or etc
- Eg - li = list('geek'), str , set(but not in order), and (not in dict)

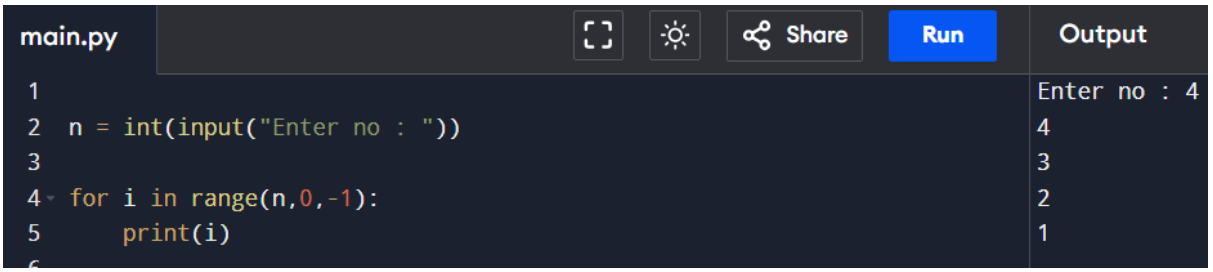
Like -

```
L = list("Wifi Learning")
print(L)
```

Output

```
['W', 'i', 'f', 'i', ' ', 'L', 'e', 'a', 'r', 'n', 'i', 'n', 'g']
```

2) How to print reverse count by for loop



range(start, stop, step)

3) How to print reverse count by While loop

code with harry

Start From Here

extra -basic

Sure 🙂 I'll explain **like you're a beginner programmer**, in **very simple language**, with **real-life comparison** and **clear examples (VS Code, Python, C++)**.

1️⃣ Code Editor – "Where you write code"

What it is (very easy):

A **code editor** is just a **text editor made for programmers**.

👉 It is the place where you **type your code**.

Real-life example:

Like **notebook + pen** where you write notes.

What it can do:

- Write code
- Color the code (syntax highlighting)
- Auto-complete words
- Find mistakes in spelling of code

Examples:

- **VS Code** ✅
- Notepad++
- Sublime Text

👉 **VS Code is mainly a Code Editor** (but can act like an IDE with extensions).

2️⃣ IDE (Integrated Development Environment) – "All-in-one tool"

What it is:

An **IDE** is a **big software** that contains:

- Code Editor
- Compiler / Interpreter
- Debugger
- Build & Run buttons

👉 Everything in **one place**.

Real-life example:

Like a **full kitchen**:

- Gas stove
- Knife
- Fridge
- Plates
  All in one room.

Why programmers use IDE:

- Easy to write
- Easy to run
- Easy to find errors
- No need to set up many tools separately

Examples:

- **PyCharm** → for Python
- **Visual Studio** → for C++, C#
- **Eclipse** → for Java

⚠️ VS Code is **not a full IDE by default**, but **becomes one** after installing extensions.

3️⃣ Compiler – "Translator for computer (whole book at once)"

What it is:

A **compiler** converts **your full code** into **machine language** (0s and 1s) **at once**.

👉 Computer does NOT understand C++ directly.

Real-life example:

Translate **whole English book** → **Hindi book** in one go.

Language that uses compiler:

- **C**
- **C++**

Example:

```
#include <iostream>
using namespace std;

int main() {
    cout << "Hello";
}
```

➡️ Compiler checks **entire program first**
➡️ If error → program will NOT run
➡️ If no error → program runs fast 🚀

4️⃣ Interpreter – "Translator line by line"

What it is:

An **interpreter** converts **code line by line**, while running.

Real-life example:

Teacher translates **one sentence at a time** while reading.

Language that uses interpreter:

- **Python**
- JavaScript

Example:

```
print("Hello")
print(10/0)   # error here
print("Bye")
```

➡️ First line runs
➡️ Error comes at second line
➡️ Program stops there

⚠️ Earlier lines already executed.

5️⃣ Debugger – "Error detective 🕵️"

What it is:

A **debugger** helps you:

- Find **where error is**
- Run code **step by step**
- Check value of variables

Real-life example:

Like **rewinding a movie scene by scene** to see where mistake happened.

What debugger can do:

- Pause program
- Check variable values
- Go line by line

Example (VS Code Debugger):

```
x = 5
y = 0
z = x / y   # debugger shows error here
```

You can:

- Stop before this line
- See `y = 0`
- Understand why error happens

🔁 Everything together (SUPER SIMPLE TABLE)

Thing	Meaning	Example
Code Editor	Write code	VS Code
IDE	Editor + Run + Debug	PyCharm
Compiler	Translate whole code	C++ compiler
Interpreter	Translate line by line	Python
Debugger	Find mistakes	VS Code debugger

🧠 How YOU actually use them

Python with VS Code:

- VS Code → Code Editor
- Python → Interpreter
- VS Code Debugger → Debugger

C++ with VS Code:

- VS Code → Code Editor
- GCC → Compiler
- VS Code Debugger → Debugger

One-line summary (remember this 👇)

Editor = write code
IDE = everything in one software
Compiler/Interpreter = convert code for computer
Debugger = find and fix mistakes

1️⃣ Compiler Languages (whole program → then run)

Meaning (1 line):

Compiler languages are converted **fully first**, then the program runs.

👉 If even **one error** → program will not run.

Common Compiler Languages:

- **C**
- **C++**
- **Rust**
- **Go**
- **Swift**
- **Objective-C**
- **Fortran**

Simple example:

int a = 10

cout << a;

❌ Missing ;
➡️ Compiler stops everything.

2️⃣ Interpreter Languages (line by line → run)

Meaning (1 line):

Interpreter languages run **one line at a time**.

👉 Error stops program **at that line only**.

Common Interpreter Languages:

- **Python**
- **JavaScript**
- **PHP**
- **Ruby**
- **Perl**
- **Bash / Shell**

Simple example:

print("Hello")

print(10/0)

print("Bye")

✔️ `"Hello"` prints
❌ Error at division
❌ `"Bye"` never prints

3️⃣ Hybrid Languages (both compiler + interpreter)

Some languages **compile first** and then **interpret at runtime**.

Hybrid Languages:

- **Java**
- **C#**
- **Kotlin**
- **Scala**

How it works:

Code → Bytecode → Run on Virtual Machine

Example:

- Java → JVM
- C# → CLR

4️⃣ Where does SQL fit? 🤔 (IMPORTANT)

Short answer:

👉 **SQL is NOT a compiler language**
👉 **SQL is NOT an interpreter language**

What SQL actually is:

SQL = Query Language

Used to:

- Read data
- Insert data
- Update data
- Delete data

From a **Database**.

5️⃣ How SQL really works (very easy)

Example SQL:

SELECT * FROM students WHERE marks > 60;

What happens internally:

1. Database (MySQL / PostgreSQL / SQL Server):
    - **Parses** the query
    - **Optimizes** it
    - **Executes** it

👉 Database engine decides:

- How to run it
- Which index to use
- How fast to fetch data

So SQL is:

✅ **Declarative language**
(You say *WHAT* you want, not *HOW* to do it)

6️⃣ SQL vs Programming Languages

Feature	SQL	Python / C++

Purpose	Work with data	Build programs
Logic flow	No loops (mostly)	Full logic
Compilation	DB handles it	Compiler / Interpreter
Use case	Data querying	Apps, software

7️⃣ Databases that use SQL

- **MySQL**
- **PostgreSQL**
- **Oracle**
- **SQL Server**
- **SQLite**

⚠️ Each database has **its own SQL engine**, but syntax is similar.

8️⃣ Very Simple Final Classification

✅ Compiler Languages

C, C++, Go, Rust

✅ Interpreter Languages

Python, JavaScript, PHP

✅ Hybrid Languages

Java, C#

✅ Query Language

SQL

One-line memory trick 🧠

- **C++** → compile → run
- **Python** → run line by line
- **Java** → compile + run on VM
- **SQL** → ask database for data

---

data type method

Yes. This is actually one of the best notes you can have while learning Python because many beginners confuse which methods belong to which data type.

Python Data Types & Their Methods (Quick Revision Notes)

> **Rule:** If a method is not listed under a data type, it **cannot** be used with that data type.

---

1. Integer (`int`)

Methods

as_integer_ratio()
bit_count()
bit_length()
conjugate()
from_bytes()
to_bytes()

---

2. Float (`float`)

Methods

as_integer_ratio()
conjugate()
fromhex()
hex()
is_integer()

---

3. Complex (`complex`)

Methods

conjugate()

---

4. Boolean (`bool`)

(Boolean is a subclass of int)

Methods

as_integer_ratio()
bit_count()
bit_length()
conjugate()
from_bytes()
to_bytes()

---

5. String (`str`)

capitalize()
casefold()
center()
count()
encode()
endswith()
expandtabs()
find()
format()
format_map()
index()
isalnum()
isalpha()
isascii()
isdecimal()
isdigit()
isidentifier()
islower()
isnumeric()
isprintable()
isspace()
istitle()

isupper()
join()
ljust()
lower()
lstrip()
maketrans()
partition()
removeprefix()
removesuffix()
replace()
rfind()
rindex()
rjust()
rpartition()
rsplit()
rstrip()
split()
splitlines()
startswith()
strip()
swapcase()
title()
translate()
upper()
zfill()

---

6. List (`list`)

append()
clear()
copy()
count()
extend()
index()
insert()
pop()
remove()
reverse()
sort()

---

7. Tuple (`tuple`)

count()
index()

---

8. Set (`set`)

add()
clear()
copy()
difference()
difference_update()
discard()
intersection()
intersection_update()
isdisjoint()
issubset()
issuperset()
pop()
remove()
symmetric_difference()
symmetric_difference_update()
union()
update()

---

9. Frozen Set (`frozenset`)

copy()
difference()
intersection()
isdisjoint()
issubset()
issuperset()
symmetric_difference()
union()

---

10. Dictionary (`dict`)

clear()
copy()
fromkeys()
get()
items()
keys()
pop()
popitem()
setdefault()
update()
values()

---

11. Bytes (`bytes`)

capitalize()
center()
count()
decode()
endswith()
expandtabs()
find()
fromhex()
hex()
index()
isalnum()
isalpha()
isascii()
isdigit()
islower()
isspace()
istitle()
isupper()
join()
ljust()
lower()
lstrip()
maketrans()
partition()
removeprefix()
removesuffix()
replace()

rfind()
rindex()
rjust()
rpartition()
rsplit()
rstrip()
split()
splitlines()
startswith()
strip()
swapcase()
title()
translate()
upper()
zfill()

---

12. Bytearray (`bytearray`)

append()
capitalize()
center()
clear()
copy()
count()
decode()
endswith()
expandtabs()
extend()
find()
fromhex()
hex()
index()
insert()
isalnum()
isalpha()
isascii()
isdigit()
islower()
isspace()
istitle()
isupper()
join()
ljust()
lower()

lstrip()
maketrans()
partition()
pop()
remove()
removeprefix()
removesuffix()
replace()
reverse()
rfind()
rindex()
rjust()
rpartition()
rsplit()
rstrip()
split()
splitlines()
startswith()
strip()
swapcase()
title()
translate()
upper()
zfill()

---

13. Range (`range`)

count()
index()

---

14. Memory View (`memoryview`)

cast()
hex()
release()
tolist()
tobytes()

Methods Comparison (Most Useful)

Method	List	Tuple	Set	Dict	String
append()	✅	❌	❌	❌	❌
extend()	✅	❌	❌	❌	❌
insert()	✅	❌	❌	❌	❌
remove()	✅	❌	✅	❌	❌
pop()	✅	❌	✅	✅	❌
clear()	✅	❌	✅	✅	❌
copy()	✅	❌	✅	✅	❌
sort()	✅	❌	❌	❌	❌
reverse()	✅	❌	❌	❌	❌
add()	❌	❌	✅	❌	❌
update()	❌	❌	✅	✅	❌
discard()	❌	❌	✅	❌	❌
union()	❌	❌	✅	❌	❌
intersection()	❌	❌	✅	❌	❌
difference()	❌	❌	✅	❌	❌
keys()	❌	❌	❌	✅	❌
values()	❌	❌	❌	✅	❌
items()	❌	❌	❌	✅	❌
get()	❌	❌	❌	✅	❌
setdefault()	❌	❌	❌	✅	❌
split()	❌	❌	❌	❌	✅
join()	❌	❌	❌	❌	✅

replace()	❌	❌	❌	❌	✅
strip()	❌	❌	❌	❌	✅
lower()	❌	❌	❌	❌	✅
upper()	❌	❌	❌	❌	✅
find()	❌	❌	❌	❌	✅
startswith()	❌	❌	❌	❌	✅
endswith()	❌	❌	❌	❌	✅

Memory Trick

- **List** → `append`, `extend`, `insert`, `sort`, `reverse`, `pop`, `remove`
- **Tuple** → `count`, `index`
- **Set** → `add`, `discard`, `remove`, `update`, `union`, `intersection`, `difference`
- **Dictionary** → `get`, `keys`, `values`, `items`, `update`, `setdefault`, `pop`, `popitem`
- **String** → Around 45+ text-processing methods such as `split`, `join`, `replace`, `strip`, `find`, `count`, `upper`, `lower`, `startswith`, `endswith`.

These notes cover the built-in methods available on Python's standard data types and are a solid quick-reference sheet for students preparing for interviews, coding tests, or day-to-day Python programming.

module - PIP class - 1

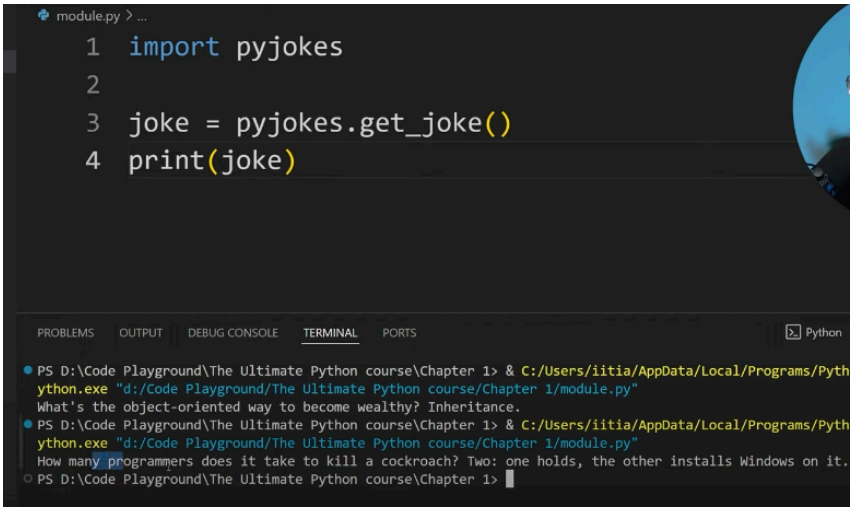
1) Module

its very simple, its a written programm or code of someone which programm or code we can easily use in our programm and also use it. By **pip install method.**
Eg - i have to find arithmetic mean so that i have to write it all programm but if someone has already made a programm for it so just i need to use their **module**, so that i can easily and hurrely find mine solution.

2) .How to install — MODULE

Step 1 - just need to write in terminal - **pip install Module Name**
Eg - pip install pyjokes

Step 2 - **import Module Name**
Eg - import pyjokes


module.py > ...
1 import pyjokes
2
3 joke = pyjokes.get_joke()
4 print(joke)
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Python
PS D:\Code Playground\The Ultimate Python course\Chapter 1> & C:/Users/iitia/AppData/Local/Programs/Python/Python
ython.exe "d:/Code Playground/The Ultimate Python course/Chapter 1/module.py"
What's the object-oriented way to become wealthy? Inheritance.
PS D:\Code Playground\The Ultimate Python course\Chapter 1> & C:/Users/iitia/AppData/Local/Programs/Python/Python
ython.exe "d:/Code Playground/The Ultimate Python course/Chapter 1/module.py"
How many programmers does it take to kill a cockroach? Two: one holds, the other installs Windows on it.
PS D:\Code Playground\The Ultimate Python course\Chapter 1>

3) REPL in Python

REPL - READ EVALUATE PRINT LOOP

4) pyttsx3 2.99 - A voice speaker module.

Step 1 - pip install pyttsx3
Step 2 -

```
import pyttsx3
engine = pyttsx3.init()

engine.say("hey how are you")
engine.runAndWait()
```

5) Imp** Empty dictionary and Empty Set - {INTERVIEW QUES}

Empty dic, S = { },
Empty set, S = ( ), # don't use s = { } as it will create empty dic.

6) split() | and how to enter two or more input in int.

7)

lib_pip_chatgpt notes

PYTHON MODULES, LIBRARIES & PIP — COMPLETE NOTES 📘

🧩 1. What is a Module?

> A **module** is a Python file (`.py`) that contains code (functions, classes, or variables) you can reuse.

✅ Example:

```
# file: mymodule.py
def greet(name):
    print(f"Hello {name}!")
```

To use it:

```
import mymodule
mymodule.greet("Nitesh")
```

🧱 2. Types of Modules

Type	Description	Example
Built-in modules	Already included in Python	`math`, `os`, `random`, `sys`, `datetime`
External modules / libraries	Installed via `pip`	`numpy`, `pandas`, `matplotlib`
User-defined modules	You create your own	`my_module.py`

🪄 3. Importing Modules

✅ Basic import:

```
import math
print(math.sqrt(25))
```

✅ Import with alias:

```
import math as m
print(m.sqrt(25))
```

✅ Import specific items:

```
from math import sqrt, pi
print(sqrt(16), pi)
```

✅ Import everything (not recommended):

```
from math import *
print(sin(pi / 2))
```

🔁 4. Re-importing a Module (after editing)

If you modify a module file and want to re-import it **without restarting** Python:

```
import importlib
importlib.reload(mymodule)
```

📦 5. What is Library in Python?

> A **library** = collection of modules designed for specific tasks.

Examples:

- `pandas` → Data analysis
- `numpy` → Mathematical operations
- `matplotlib` → Data visualization
- `requests` → API / web requests

So:
Module = single file
Library = collection of modules

⚙️ 6. What is pip?

`pip` = Python's **Package Installer**
Used to install, upgrade, and remove external libraries.

💻 7. Most Useful `pip` Commands

Command	Description
`pip --version`	Check pip version
`pip install <package>`	Install a package
`pip install <package>==1.2.3`	Install specific version
`pip install --upgrade <package>`	Upgrade to latest version
`pip uninstall <package>`	Remove a package
`pip show <package>`	Show package info (version, location)
`pip list`	Show all installed packages
`pip freeze`	Show all packages (used for requirements.txt)
`pip install -r requirements.txt`	Install all packages listed in file

🧾 8. Checking Library Version (3 Ways)

```
import pandas
print(pandas.__version__)   # ✅ Method 1
```

OR

```
pip show pandas          # ✅ Method 2
```

OR

pip list | find "pandas" # ✅ Method 3 (Windows)

🧰 9. Where Packages Are Installed

Run this in Python:

```
import sys
print(sys.path)
```

→ It shows directories where Python looks for modules.

🧑‍💻 10. Creating Your Own Module

Step 1: Create `mycalc.py`

```
def add(a, b):
    return a + b
```

Step 2: Use it

```
import mycalc
print(mycalc.add(5, 10))
```

🧠 Trick: Python will search `mycalc.py` in the same folder or system path.

🗂️ 11. Package vs Module

Feature	Module	Package
File type	Single `.py` file	Folder with many modules
Example	`math.py`	`numpy/`, `pandas/`
Has `__init__.py`?	No	Yes (acts as entry point)

Example structure:

```
myproject/
│
├── shapes/
│   ├── __init__.py
│   ├── circle.py
│   └── square.py
└── main.py
```

🌍 12. Virtual Environment (Must-Know for Projects)

💬 Why use it?

- To isolate your project's dependencies.
- Prevent version conflicts between projects.

✅ Create and use venv:

python -m venv env

Activate:

- **Windows:** `env\Scripts\activate`
- **Mac/Linux:** `source env/bin/activate`

Install packages inside it:

pip install numpy

Deactivate:

deactivate

🚀 13. requirements.txt File

Used to share your project dependencies.

Create it:

pip freeze > requirements.txt

Install everything from it:

pip install -r requirements.txt

🧩 14. Common Built-in Modules to Know

Category	Module	Usage
Math	`math`, `random`, `statistics`	Calculations
OS / System	`os`, `sys`, `platform`, `shutil`	File operations, system info
Date/Time	`datetime`, `time`	Date & time
Web	`requests`, `urllib`	Web access
Data	`json`, `csv`, `pickle`	Data handling
Debugging	`logging`, `traceback`	Logs, errors
Others	`itertools`, `functools`, `collections`	Advanced tools

🧠 15. Pro Tips & Best Practices

✅ Always use `import <module>` at the top of the file
✅ Use **aliases** (short names) for readability → `import numpy as np`
✅ Avoid `from module import *` (can cause name conflicts)
✅ Keep a `requirements.txt` for every project
✅ Use virtual environments to isolate dependencies
✅ Upgrade pip regularly → `python -m pip install --upgrade pip`
✅ Use `help(module)` or `dir(module)` to explore any library
✅ For documentation → https://pypi.org

🪄 16. Memory Trick (for Revision)

M-I-P formula:

- **M** → Module → single file
- **I** → Import → use it
- **P** → pip → install it

So:

"If it's not built-in → pip install it → import it → enjoy it!" 😎

💡 17. Quick Practice Questions (with hints)

1. Install and import `requests` and print its version.
2. Create your own module that adds, subtracts, multiplies.
3. Use `pip freeze` and explain what the output means.
4. Find where Python stores installed packages.
5. Create a virtual environment and install `pandas` only there.

ques, crt env

1. **Create a virtual environment and install `pandas` only there.**
2. **Use `pip freeze` and explain what the output means.**
3. **Use Requirement.txt**
4. **And deactivate env**

---

1. How to create `virtual environment` and install panda only.

First **activate your virtual environment**:

Windows:

env\Scripts\activate

Then **Install Pandas.**

Pip install pandas

Then can be use – import pandas

✅ 1. How to use `pip freeze`

First **activate your virtual environment**:

Windows:

env\Scripts\activate

Mac/Linux:

source env/bin/activate

Now run:

pip freeze

✅ 2. What does `pip freeze` output mean?

`pip freeze` shows **all installed packages in the current virtual environment** along with their **exact versions**, in this format:

package_name==version_number

Example (after installing only pandas):

pandas==2.2.2

numpy==1.26.4

python-dateutil==2.8.2

pytz==2024.1

six==1.16.0

✔️ Why does this happen if we installed only *pandas*?

Because **pandas depends on other packages**, so pip automatically installs:

- numpy
- python-dateutil
- pytz
- six

These are called **dependencies**.

👉 `pip freeze` shows **all packages that are installed inside your env**, including dependencies.

✅ 3. How to relate `pip freeze` to `requirements.txt`

`requirements.txt` is a file in Python projects that lists the **exact versions** of packages needed to run the project.

You create it using:

pip freeze > requirements.txt

This means:

- Whatever is shown in `pip freeze`
- Will be written into the `requirements.txt` file

Example `requirements.txt` (auto-generated):

pandas==2.2.2

numpy==1.26.4

python-dateutil==2.8.2

pytz==2024.1

six==1.16.0

✔️ Purpose of this file:

- Anyone can recreate **the same environment** by running:

pip install -r requirements.txt

This installs **exact same versions** → avoids errors and version conflicts.

4. How to Deactivate -

deactivate
Or
Deactivate

📝 Final Explanation (You can write this in your assignment):

pip freeze lists all packages installed in the current virtual environment along with their exact versions.
This output is used to generate a `requirements.txt` file, which stores the dependencies of a project.
The file is created with the command:

pip freeze > requirements.txt

This allows other developers to recreate the same environment using:

pip install -r requirements.txt

This ensures the project runs the same everywhere.

Interpreter

⭐ VS Code – Python Interpreter Notes (Best & Complete Notes)

🔶 1. What is a Python Interpreter?

A **Python interpreter** is the program that actually *runs your Python code*.
Examples:

- `python.exe` (Windows)
- `python3` (Linux/Mac)
- `env/Scripts/python.exe` (Virtual Environment)

Every project can use a **different interpreter**.

🔶 2. Types of Python Interpreters

VS Code can detect multiple interpreters:

✔️ (1) System Python

Installed globally in your system, example:

C:\Users\DELL\AppData\Local\Programs\Python\Python312\python.exe

✔️ (2) Virtual Environment Interpreter

Each virtual environment has its **own Python**, example:

project/env/Scripts/python.exe

✔️ (3) Conda Environment Interpreter

If someone uses Anaconda, each conda env has its own interpreter.

🔶 3. Why Interpreter Selection Matters

Choosing the right interpreter ensures:

✔️ Correct packages

Your code runs using the packages installed in that interpreter.

✔️ Correct environment

You avoid errors like:

- “ModuleNotFoundError”
- “ImportError”
- Code running with wrong package versions

✔️ VS Code auto-activation

VS Code terminal and debugger use the interpreter you select.

🔶 4. How to Select Interpreter in VS Code

Method 1: Command Palette (Recommended)

1. Press **CTRL + Shift + P**
2. Type: `Python: Select Interpreter`
3. VS Code will show a list:

Example list:

Python 3.12.3 (system)
Python 3.10 (conda)
Python 3.12.3 ('env': venv)
Python 3.8 ('myproject_env': venv)

4. Choose the correct one.

Method 2: Using the Status Bar

Bottom-right corner of VS Code shows the active interpreter:

Python 3.12.3 ('env')

Click it → interpreter selection menu opens.

🔶 5. How VS Code Chooses Interpreter Automatically

VS Code looks for:

- `.venv`
- `env`
- `venv`
  inside your project folder.

If found → VS Code automatically selects it.

This is why sometimes after creating a venv, VS Code automatically activates it.

🔶 6. How Interpreter Affects Virtual Environments

✔️ If you select the env interpreter:

- VS Code terminal auto-activates venv

- `pip install` installs inside env
- Running code uses env libraries
- `pip list` shows only env packages

✔️ If you select system interpreter:

- Packages install globally
- venv is NOT used
- Code outside env may run differently

✔️ If interpreter is incorrectly selected:

- VS Code may use wrong pip and wrong python

🔶 7. How to Check Which Interpreter is Active

In VS Code Terminal:

where python
where pip

If paths contain your venv folder:

.../env/Scripts/python.exe

➡️ You're inside the virtual environment.

If paths contain system folder:

.../Python312/python.exe

➡️ You're using system Python.

🔶 8. Common Problems & Solutions

❌ Problem 1: “pip install” not working

Reason: Wrong interpreter selected
Fix: Change interpreter → select venv

❌ Problem 2: import error like

ModuleNotFoundError: No module named 'pandas'

Reason: Code using system Python, but pandas installed in venv
Fix: Select venv interpreter

❌ Problem 3: `pip freeze` shows no packages

Reason: VS Code automatically uses system Python
Fix: Switch interpreter to env

🔶 9. Best Practices for Projects

✔️ Always create a venv inside project:

python -m venv env

✔️ Always select interpreter after creating venv

(Use the bottom-right corner of VS Code)

✔️ Use `requirements.txt` for sharing project:

pip freeze > requirements.txt

✔️ If you open a new folder in VS Code → reselect interpreter

(VS Code does not remember interpreters per system, only per workspace)

🔶 10. Quick Summary (For Exam / Notes)

- A Python interpreter is the program that runs Python code.
- Each venv has its own interpreter.
- VS Code uses the interpreter you select to run & debug Python.
- Wrong interpreter → wrong pip, wrong imports, errors.
- Use **Python: Select Interpreter** from the command palette.
- Always select the venv interpreter for project isolation.
- Use `where python` and `where pip` to check active interpreter.
- `pip freeze` output depends *on the interpreter you selected*.

input / Split / Map

How to take too many inputs -

✅ 1. Take inputs separately

```
a = input("Enter first value: ")
b = input("Enter second value: ")
c = input("Enter third value: ")

print("You entered:", a, b, c)
```

✅ 2. Take multiple inputs in one line (space-separated)

```
x, y, z = input("Enter three values separated by space: ").split()

print("You entered:", x, y, z)
```

👉 Example: If you type

10 20 30

Output will be:

You entered: 10 20 30

✅ 3. If you want numbers instead of strings

```
p, q, r = map(int, input("Enter three numbers: ").split())

print("Sum =", p + q + r)
```

🔹 1. Enter many values in one line (space-separated)

```
nums = list(map(int, input("Enter numbers separated by space: ").split()))
print("You entered:", nums)
```

👉 Example input:

5 10 15 20 25

👉 Output:

You entered: [5, 10, 15, 20, 25]

🔹 2. Enter many values line by line until a condition (e.g. type 'done')

```
data = []
while True:
    val = input("Enter value (or type 'done' to stop): ")
    if val.lower() == "done":
        break
    data.append(val)

print("All inputs:", data)
```

👉 Example:

```
Enter value (or type 'done' to stop): apple
Enter value (or type 'done' to stop): mango
Enter value (or type 'done' to stop): banana
Enter value (or type 'done' to stop): done
All inputs: ['apple', 'mango', 'banana']
```

🔹 3. Enter a fixed count of inputs (like 10 values)

```
n = int(input("How many numbers do you want to enter? "))
values = []

for i in range(n):
    val = int(input(f"Enter number {i+1}: "))
    values.append(val)

print("Your numbers:", values)
```

✅ So you can choose depending on your case:

- **One line space-separated** → `split()`

- **Until stop word** → `while True`
- **Fixed number** → `for loop`

Good question 👍 Let's break it down simply.

🔹 What is `map()` in Python?

The `map()` function applies another function to **every item** in an iterable (like a list, tuple, or input split).

👉 General form:

map(function, iterable)

- `function` → a function you want to apply (like `int`, `str.upper`, `float`, etc.)
- `iterable` → something that contains items (like `list`, `tuple`, `input().split()`)

It returns a **map object** (which is an iterator), so we usually convert it into a list, tuple, or set.

🔹 Example 1: Convert string numbers to integers

nums = list(map(int, ["10", "20", "30"]))

print(nums)

👉 Output:

[10, 20, 30]

Here, `map(int, ["10","20","30"])` means:

- take each element of `["10","20","30"]`
- apply `int()` on it → `[10, 20, 30]`

🔹 Example 2: Square every number

```
def square(x):
    return x * x

numbers = [1, 2, 3, 4]
squares = list(map(square, numbers))

print(squares)
```

👉 Output:

[1, 4, 9, 16]

🔹 Example 3: With `input().split()`

When you write:

```
nums = list(map(int, input("Enter numbers: ").split()))
```

- `input().split()` → takes input like `"5 10 15"` and makes it `["5","10","15"]`
- `map(int, ["5","10","15"])` → converts all strings to integers `[5,10,15]`

- `list(...)` → stores them in a list.

👉 In short:
`map()` = "apply a function to every element in a sequence."

Badiya — chalo clear aur *kabhi na bhoolne* waale style mein samjhate hain. Hinglish mein — simple, visual, aur examples ke saath. 🚀

1) `.split()` — exact definition + easy image

Definition (simple):
`.split()` ek string method hai jo string ko chhote tukdo (tokens) mein **baant** deta hai aur un tukdon ko **list** bana ke deta hai.

Socho: string = ek roti, `.split()` = chaku — roti ko tukdo mein kaat do.

Default behavior: agar aap `.split()` bina argument ke use karo → woh **saare whitespace** (space, tab, newline) ko treat karke **extra spaces hata ke** split karta hai.

```
s = "  hello   world  "

print(s.split())   # -> ['hello', 'world']
```

Custom separator / maxsplit:

```
"1,2,3".split(',')          # -> ['1','2','3']

"a b c d".split(maxsplit=2)  # -> ['a','b','c d']
```

Important pitfall:
`.split(' ')` (explicit space) *alagalag behave karega* — multiple spaces ko empty strings bana deta hai:

```
" a  b ".split()    # -> ['a','b']

" a  b ".split(' ') # -> ['', 'a', '', 'b', '']
```

2) `map()` — exact definition + easy image

Definition (simple):
`map(function, iterable)` — iterable ke **har item** par `function` **apply** karta hai aur result deta hai. Python 3 mein result **iterator (map object)** hota hai — isse list/tuple mein convert karna padta hai agar aap baar-baar use karna chahte ho.

Socho: `map` ek paintbrush jaisa hai — har item ko ek kaam (paint/function) lagaa de.

strings ko numbers mein convert karna:

tokens = ["10","20","30"]

nums = list(map(int, tokens)) # -> [10, 20, 30]

Lazy / one-pass:
`m = map(int, ["1","2","3"])`
`print(list(m))` -> `[1,2,3]`
`print(list(m))` -> `[]` (kyunki iterator ek baar khatam ho gaya)

Lambda / multiple iterables:

lambda example

list(map(lambda x: x*x, [1,2,3])) # -> [1,4,9]

multiple iterables (pairwise)

list(map(lambda x,y: x+y, [1,2], [10,20])) # -> [11,22]

3) `.split()` + `map()` — kyun dono bahut common hain?

User input vaakya: `"10 20 30"` — program ko numbers chahiye → pehle `.split()` se tokens banao, phir `map(int, ...)` se convert karo.

```
nums = list(map(int, input().split()))
```

Equivalent long form (same kaam, lekin lamba):

```
s = input()

parts = s.split()

nums = []

for p in parts:

    nums.append(int(p))
```

Conclusion: Ha, bina `split()` aur `map()` ke code likh sakte ho — par vo verbose, zyada lines aur error-prone hoga. `split()` aur `map()` se code chhota, readable aur common tasks (parsing, converting) asaan ho jaate hain.

4) `map()` vs list comprehension — kab kya use karein?

```
# map (built-in function fast for simple funcs)

list(map(int, tokens))


# list comprehension (zyaada flexible, pythonic for complex logic)

[int(x) for x in tokens if x != ""]
```

Rule of thumb: agar simple built-in function apply karna hai (int, str, float, str.upper) → `map` perfect. Agar condition, expression, ya complex logic → list comprehension zyada readable.

5) Memory tricks (Kabhi na bhoolne waale lines)

- `.split()` = **"chaku"** — string ko tukdo mein kaat do.
- `map()` = **"paintbrush"** — har tukde par same kaam chala do.
- Yaad rakho: `split()` returns **list**, `map()` returns **iterator** (Python 3) — isliye `list(map(...))` common pattern hai.

6) Common mistakes & fixes

- `nums = map(int, input().split())` → agar aap `print(nums)` karoge to `<map object ...>` dikhega. Fix: `list(nums)` ya `for x in nums:`.
- Extra spaces? use `.split()` (no arg), na ki `.split(' ')`.
- CSV mein quoted commas ko handle karne ke liye `.split(',')` se dikkat aa sakti hai — use `csv` module.

7) Short practice (try karo) — inputs + expected outputs

1. Input: `10 20 30`
   Code:

```
nums = list(map(int, input().split()))

print(sum(nums))
```

Output: `60`

2. Input: `alice,bob,charlie`
   Code:

```
names = input().split(',')

print(list(map(str.title, names)))
```

Output: `['Alice', 'Bob', 'Charlie']`

3. Convert string of pairs to sums: Input: `1-2 3-4 5-6`
   Code:

```
pairs = input().split()

sums = [sum(map(int, p.split('-'))) for p in pairs]

print(sums)
```

If input `1-2 3-4 5-6` → Output: `[3, 7, 11]`

8) Final 1-line takeaway (imperative)

- `.split()` = string ko token mein baant do.
- `map(f, seq)` = har token par function `f` lagao.
- `list(map(...))` agar tumhe list chahiye aur `map` ka result baar-baar use karna hai.

Agar chaho toh ab main 5 quick exercises banake doonga (answers ke saath) taaki tum practice kar sako — bol do agar chahiye. 😊

class - 2

- Factorial (recursion) with functions -

```
def fact(n):
    if (n == 1 or n == 0):
        return 1
    return n * fact(n-1)

n = int(input("Enter no for factorial : "))
print(f"factorial of {n} is {fact(n)}")
```

- Return in Function

Its used in to assign in value in something so that we can use function value in other functions or in other definition.
Like -

```
def func1(num1,num2 =,num3 = 6):
    avg = (num1+num2+num3)/3
    return avg

a = func1(4,5)

b = 8

print(a*b)
```

- Use of this - end=" " - use for join two print in one line

When you write,
Print (“Radha”)
Print (“krishna”)
It will give **output**
Radha
Krishna

But,

By using this - **end=" "**, **its = “ “ separation**, as much you will leave space you will get that much space

You will get

```
print("Radha",end=" ")
print("Krishna",end=" ")
```

Output - Radha Krishna

🔹 Closure in Python

1) Core Idea

Closure = ek **function ke andar ek aur function**, jo **outer function ke variables ko yaad rakhta hai** (even after outer function khatam ho jaye).

Matlab: inner function = data + code dono ko carry karta hai.

---

2) Simple Example (without closure)

```
def outer():
    x = 10
    def inner():
        print(x)
    inner()

outer()
```

✅ Output: `10`

Here, `inner()` sirf tab chalega jab `outer()` chalega. Baad me direct use nahi kar sakte.

---

3) Closure Example

```
def outer():
    x = 10
    def inner():
        print(x)
    return inner   # return function itself, not call

my_func = outer()   # outer runs, returns inner
my_func()           # call inner later
```

✅ Output: `10`

👉 Yaha outer function khatam ho gaya, lekin `inner` function ke paas abhi bhi `x=10` stored hai.
Ye hi **closure** hai.

---

4) Why use Closure?

- **Encapsulation**: data ko hide karke sirf ek function se access karna.
- **Stateful functions**: function ke andar values yaad rakhna without global variables.
- **Callbacks / decorators**: closures ka base decorators hote hain.

---

5) Without Closure vs With Closure

❌ Without closure (using global var)

```
x = 10
def inner():
    print(x)

inner()
```

Problem: `x` global hai → koi bhi change kar sakta hai, unsafe.

✅ With closure

```
def outer():
    x = 10
    def inner():
        print(x)
    return inner

f = outer()
f()
```

Benefit: `x` protected hai, sirf `inner()` use kar sakta hai.

---

6) Closure with State (like counter)

```
def make_counter():
    count = 0
    def counter():
        nonlocal count
```

```
        count += 1
        return count
    return counter

c = make_counter()
print(c())  # 1
print(c())  # 2
print(c())  # 3
```

👉 Yaha `count` variable normal function ke baad bhi "alive" hai — because of closure.

---

7) Real life example: multiplier

```
def make_multiplier(n):
    def multiplier(x):
        return x * n
    return multiplier

double = make_multiplier(2)
triple = make_multiplier(3)

print(double(5))  # 10
print(triple(5))  # 15
```

Benefit: Reusable functions with custom state.

---

8) Closure vs Class

Closures behave like **lightweight classes**.
For example:

With Class:

```
class Counter:
    def __init__(self):
        self.count = 0
    def __call__(self):
        self.count += 1
        return self.count

c = Counter()
print(c())  # 1
```

```
print(c())  # 2
```

With Closure (shorter way):

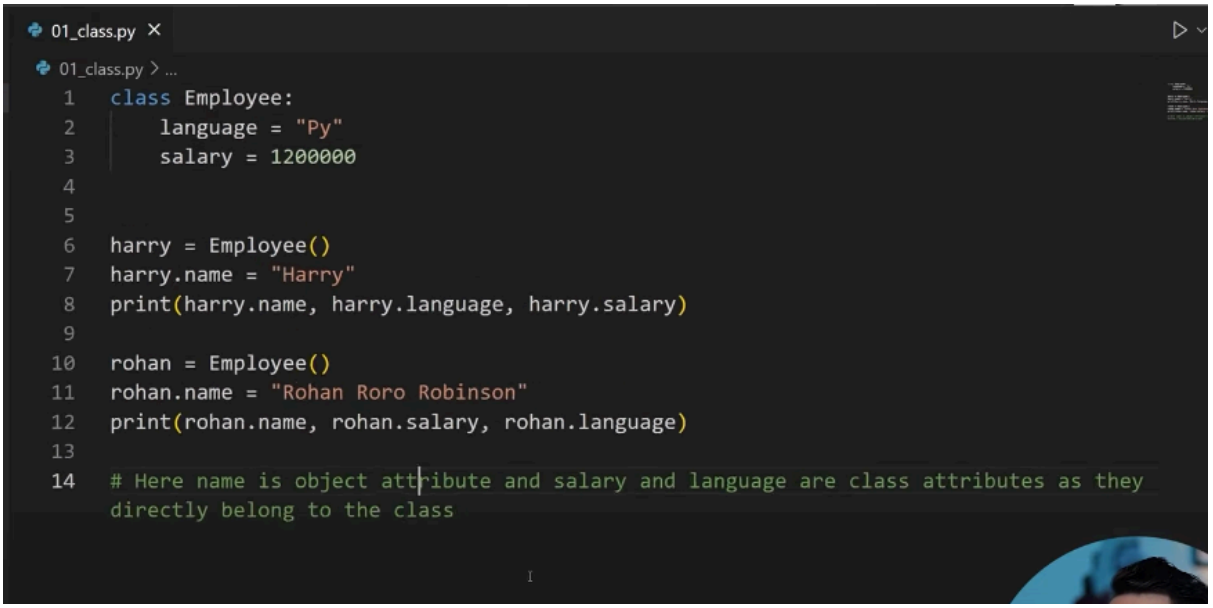
```
def make_counter():
    count = 0
    def counter():
        nonlocal count
        count += 1
        return count
    return counter

c = make_counter()
print(c())  # 1
print(c())  # 2
```

🔑 Summary (remember forever)

- **Closure = function + environment (outer variables)**.
- Outer function execute hote hi variables destroy nahi hote, agar inner function return ho gaya hai.
- Useful for hiding data, making stateful functions, decorators.
- Agar tumhe **"function ke saath data pack karna hai"** but class banana heavy lag raha hai → closure is perfect.

class - 3

Object attribute (also knowns as Instance attribute) and Class attribute.


01_class.py
01_class.py > ...
1 class Employee:
2 language = "Py"
3 salary = 1200000
4
5
6 harry = Employee()
7 harry.name = "Harry"
8 print(harry.name, harry.language, harry.salary)
9
10 rohan = Employee()
11 rohan.name = "Rohan Roro Robinson"
12 print(rohan.name, rohan.salary, rohan.language)
13
14 # Here name is object attribute and salary and language are class attributes as they directly belong to the class

__init__ method (dunder method).

__ = dunder method,
And not all dunder method automatically call, just only __init__ method call itself automatically.

One smart way to write import…

1) **Write import random**
   And **further** where we use it write there-
   random.randint

```python
import random

class Train:
    def book(self, trainNo, fro, to):
        print(f"Ticket is booked in train no: {trai
        from {fro} to {to}")

    def getStatus(self, trainNo):
        pass

    def getFare(self, trainNo, fro, to):
        print(f"Ticket fare in train no: {trainNo}
        {fro} to {to} is {random.randint(222, 5555
```

2) Just write -
From random import randint(where we have to use this)
And in further -
Not need to write random.randint - just leave randint.

```python
from random import randint

class Train:
    def book(self, trainNo, fro, to):
        print(f"Ticket is booked in train no: {trainNo}
        from {fro} to {to}")

    def getStatus(self, trainNo):
        pass

    def getFare(self, trainNo, fro, to):
        print(f"Ticket fare in train no: {trainNo} f
        {fro} to {to} is {randint(222, 5555)}")
```

file_IO

```
'''1. open
f = open("cwh_chapters/ch_9_FILE_IO/poems.txt","r")
conet = f.read()
print(f)
f.close()
'''

''' 2. write() -- create new file
f = "Hey how are you Buddy!!!"
with open("new_cerrt.txt","w") as crt:
    print(crt.write(f))

#or 2nd way for in this folder...

f = "hey how are you Buddy\n they are the our bros\n who are they"
with open("cwh_chapters/ch_9_FILE_IO/newlog.txt","w") as k:
    k.write(f)
'''

'''3. clear words of file
with open("new_cerrt.txt","w") as de:
    de.write("")
'''

'''4. delete that file
import os
#os.remove("new_cerrt.txt")

#second way
file_name = "new_crt.txt"
if (os.path.exists(file_name)):
    os.remove(file_name)
    print("File successfully deleted")
else:
    print("File not found")
'''

''' 5. readlines() -- read all line by line and make this in a
lists...
```

```
with open("cwh_chapters/ch_9_FILE_IO/p.txt","r") as f:
    a = f.readlines()
print(a)
print(type(a)) #- it will show lists

#or find any words in that file

with open("cwh_chapters/ch_9_FILE_IO/p.txt","r") as k:
    a = k.readlines()
if("love" in a):
    print("Yes")
else:
    print("No")
'''

''' 6. readline() -- read only first line or after that one
line...
with open("cwh_chapters/ch_9_FILE_IO/p.txt","r") as f:
    print("1st line",f.readline())
    print("2nd line", f.readline())
    print("3rd line",f.readline())
    print("4th line",f.readline())
    print("5th line",f.readline())
    print("6th line",f.readline()) #--- this line will show blank
as all line already printed

#other method

with open("cwh_chapters/ch_9_FILE_IO/p.txt","r") as f:
    a = f.readline()
    while (a != ""):
        print(a)
        a = f.readline()
or

with open("cwh_chapters/ch_9_FILE_IO/p.txt","r") as f:
    for line in f:
        print(line)
'''

''' 7.write
```

```
with open("cwh_chapters/ch_9_FILE_IO/newlog.txt","w") as h:
    h.write("hey Nitesh is come here\nwooooooooo!!!!!\nWhat again
wow!!\nNo i can't trust it...")
'''

''' 8. append {a} -- its like just append new lines with exists
ones
with open("cwh_chapters/ch_9_FILE_IO/newlog.txt","a") as h:
    h.write("I am sure\n just this all depend on you\nlike are you
trusting me or not...")
'''

''' 9. +  - open for updating.
"w+" → clears file first, then writes new content.
"a+" → keeps old content, adds new at the end.

by a+
with open("cwh_chapters/ch_9_FILE_IO/newlog.txt","a+") as h:
    h.write("\nthis is new \n just Sentence\nlike are you trusting
me or not...")

by w+
with open("cwh_chapters/ch_9_FILE_IO/newlog.txt","w+") as h:
    h.write("\nthis is new \n just Sentence\nlike are you trusting
me or not...")
'''

'''some extras - thasa
with open("cwh_chapters/ch_9_FILE_IO/pk.txt","w") as k:
    k.write("")

del --

import os
os.remove("cwh_chapters/ch_9_FILE_IO/pk.txt")
'''
```

short_notes oops

OOP in Python — Ultimate Hinglish Notes

---

1. What is OOP?

👉 Programming style jisme hum **real-world entities** ko code me represent karte hain as **classes & objects**.

- **Class = Blueprint / Design**
- **Object = Actual instance of that design**

💡 Real world: Mobile Model = Class, Tumhare paas actual iPhone 15 = Object.

If not OOP → Har feature ke liye alag-alag function likhne padte, sab scattered hote, reuse aur management mushkil.
With OOP → Everything grouped, reusable, structured.

---

2. Class & Object

```
class Student:
    def __init__(self, name, roll):
        self.name = name
        self.roll = roll

s1 = Student("Nitesh", 53)
```

- **Class** → defines structure (what data + what behavior).
- **Object** → real data based instance.

If no class → Har student ko store karne ke liye alag dicts banate, koi order nahi.
With class → Neat, reusable, easy to manage.

---

3. Encapsulation (lapetna)

👉 Bundle **data + methods** together in one unit (class).

Without Encapsulation →

- Variables idhar, functions udhar → messy
- Data easily accessible (less security)

With Encapsulation →

- Data + methods ek jagah
- Access modifiers possible:
  - Public: `self.name`
  - Protected: `self._name`
  - Private: `self.__name`

💡 Think mobile: apps + data ek hi box me, scattered nahi.

---

4. Abstraction (hide details)

👉 Show only **what is necessary**, hide complex internal working.

```
class Car:
    def start(self):
        self.__ignite()
        print("Car started")

    def __ignite(self):
        print("Engine ignition")
```

Without abstraction → User could directly call internal methods (`__ignite`) and break flow.
With abstraction → User ke liye simple interface → `car.start()`.

💡 Real world: ATM → just withdraw option, andar note counting kaise hoti hai user ko nahi pata.

---

5. Inheritance (reuse / virasat)

👉 Ek class dusri class ki properties/methods inherit kare.

```
class Animal:
    def sound(self):
        print("Some sound")

class Dog(Animal):
    def sound(self):
        print("Bark")
```

Without inheritance → Repeated code, har class me bar bar same logic.
With inheritance → Code reuse, DRY (Don't Repeat Yourself).

💡 Example: All vehicles have wheels. Car, Bike, Truck → inherit wheel property, aur apna extra add karte.

---

6. Polymorphism (many forms)

👉 Same name, different behavior depending on context.

```
class Bird:
    def sound(self):
        print("Chirp")

class Crow(Bird):
    def sound(self):
        print("Kaw Kaw")
```

Without polymorphism → Har type ke liye alag-alag function call karna padta (crow_sound, parrot_sound...).
With polymorphism → Common interface → `bird.sound()`, object ke hisaab se response change.

💡 Example: Same word "run" → program run, athlete run, machine run → context alag, action alag.

---

7. Constructor (`__init__`)

👉 Special method, auto-called jab object create hota hai.

Without constructor → Object banane ke baad manually set karna padta.
With constructor → Init time pe hi values set.

```
class Student:
    def __init__(self, name):
        self.name = name
```

---

8. `self`

👉 Reference to current object.

Without self → Python ko pata hi nahi chalega ki kaunse object ka variable update karna hai.
With self → Object-specific data store possible.

---

9. Class vs Instance variables

- **Class var** → Shared by all objects.
- **Instance var** → Unique for each object.

Without class vars → Har object me duplicate data store.
With class vars → Memory efficient.

---

10. Special methods (`__str__`, etc.)

- `__str__` → controls how object prints.
- `__len__`, `__add__`, etc. → dunder methods for custom behavior.

Without these → Object print karega kuch random memory address.
With these → Human readable, usable.

🎯 Final Golden Recap

- **Class** = design, **Object** = product
- **Encapsulation** = data + methods ek jagah
- **Abstraction** = sirf zaroori cheez dikhana
- **Inheritance** = reuse, virasat
- **Polymorphism** = ek naam, multiple behavior
- **self** = current object reference
- **constructor** = object bante hi auto init

💡 Core Idea:
Without OOP → messy, repetitive, hard to scale.
With OOP → structured, reusable, maintainable, real-world like.

oops

OOP Notes — code examples ke saath (hinglish)

---

1) Class & Object — core idea

Core: Class = blueprint; Object = real instance.

Without OOP (procedural):

```
# storing students as dicts and functions operating on them
students = []

def add_student(name, roll):
    students.append({"name": name, "roll": roll})

def print_students():
    for s in students:
        print(s["name"], s["roll"])

add_student("A", 1)
add_student("B", 2)
print_students()
```

Problem: Data + behavior separate → logic scattered, harder to maintain and reuse.

With OOP:

```
class Student:
    def __init__(self, name, roll):
        self.name = name
        self.roll = roll

    def show(self):
        print(self.name, self.roll)

s1 = Student("A", 1)
s2 = Student("B", 2)
s1.show(); s2.show()
```

Benefit: Data + behaviour ek jagah; reusability and clarity.

---

2) `__init__` (constructor) & `self`

Core: `__init__` initializes object; `self` refers to current instance.

Without constructor:

```
class Car:
    pass

c = Car()
c.speed = 0   # manual setup every time
```

Problem: Every object must be initialized manually — error-prone.

With constructor + self:

```
class Car:
    def __init__(self, model):
        self.model = model
        self.speed = 0

c = Car("Honda")
print(c.model, c.speed)
```

Benefit: Objects always start in valid state.

---

3) Encapsulation (hide internal data / protect invariants)

Core: Bundle data+methods; prevent external code from breaking internal state.

Without encapsulation (bad):

```
class BankAccount:
    def __init__(self, bal):
        self.balance = bal

acct = BankAccount(1000)
acct.balance = -10000   # external code can break balance
```

Problem: External code can set invalid states.

With encapsulation (good):

```
class BankAccount:
    def __init__(self, bal):
        self.__balance = bal      # private by name-mangling

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
        else:
            raise ValueError("Invalid withdraw")

    def get_balance(self):
        return self.__balance

acct = BankAccount(1000)
acct.deposit(500)
print(acct.get_balance())   # 1500
# acct.__balance  # AttributeError (protected)
```

Notes: `__name` causes name-mangling (`_ClassName__name`) — not absolute private but prevents accidental access.

Benefit: Class controls valid operations; invariants maintained.

---

4) Abstraction (simple public interface)

Core: User sees only necessary methods; internal details hidden.

Without abstraction:

```
class Car:
    def check_injectors(self): pass
    def ignite(self): pass
    def adjust_timing(self): pass

c = Car()
# user must know to call sequence: check_injectors -> ignite -> adjust_timing
```

Problem: User must know internal sequence; misuse possible.

With abstraction:

```
class Car:
    def start(self):
        self.__check_injectors()
        self.__ignite()
        print("Car started")

    def __check_injectors(self): pass
    def __ignite(self): pass

c = Car()
c.start()   # simple, safe interface
```

Benefit: Single simple call; internal complexity hidden & controlled.

---

5) Inheritance (code reuse)

Core: Child classes reuse/extend parent behavior.

Without inheritance (duplication):

```
class Dog:
    def sound(self): print("Bark")
class Cat:
    def sound(self): print("Meow")
# if many animals, duplicate structure
```

With inheritance:

```
class Animal:
    def sound(self):
        raise NotImplementedError

class Dog(Animal):
    def sound(self): print("Bark")

class Cat(Animal):
    def sound(self): print("Meow")
```

Benefit: Shared logic in parent; less duplication. **Pitfall:** deep/complex hierarchies can be fragile — prefer composition sometimes.

---

6) Polymorphism (same interface, different behavior)

Core: Same method name works for different types.

Example:

```
class Bird:
    def fly(self): print("generic fly")

class Sparrow(Bird):
    def fly(self): print("sparrow fly")

class Airplane:
    def fly(self): print("airplane fly")  # not inherited but same method name

for f in [Sparrow(), Airplane()]:
    f.fly()   # polymorphism via duck typing
```

Benefit: Generic code can call `obj.fly()` regardless of type.

---

7) Class variables vs Instance variables

Core: class vars shared by all instances; instance vars unique.

Gotcha (mutable class var trap):

```
class Bag:
    items = []  # class variable (shared)

a = Bag(); b = Bag()
a.items.append(1)
print(b.items)  # [1]  -> unexpected shared state
```

Correct (instance var):

```
class Bag:
    def __init__(self):
        self.items = []   # instance variable

a = Bag(); b = Bag()
a.items.append(1)
print(b.items)  # []
```

Benefit: Avoid subtle shared-state bugs.

---

8) `@staticmethod`, `@classmethod`, and `@property`

Core: Utility methods and alternative constructors; property to access like attribute.

Examples:

```
class Circle:
    pi = 3.14

    @staticmethod
    def area_by_radius(r):
        return Circle.pi * r * r

    @classmethod
    def from_diameter(cls, d):
        return cls(d/2)

    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        return self._radius

    @radius.setter
    def radius(self, r):
        if r <= 0: raise ValueError
        self._radius = r
```

Benefit: Clean API (`Circle.from_diameter(10)`), utility with no instance, safe attribute access via property.

---

9) Dunder methods (special methods / operator overloading)

Core: Make objects behave like builtins (`+`, `str()`, `len()`).

Example:

```
class Vector:
```

```
    def __init__(self, x, y):
        self.x, self.y = x, y
    def __add__(self, other):
        return Vector(self.x+other.x, self.y+other.y)
    def __str__(self):
        return f"Vector({self.x},{self.y})"

v1 = Vector(1,2)
v2 = Vector(3,4)
print(v1 + v2)   # Vector(4,6)
```

Benefit: Natural, readable object usage; helpful for debugging and tests.

---

10) Composition vs Inheritance (has-a vs is-a)

Core: Prefer composition (has-a) when relationship is "uses" not "is".

Inheritance misuse example:

```
class Engine:
    def start(self): pass

class Car(Engine):   # wrong: Car is not a type of Engine
    pass
```

Composition (correct):

```
class Engine:
    def start(self): print("engine started")

class Car:
    def __init__(self):
        self.engine = Engine()
    def start(self):
        self.engine.start()
        print("car started")
```

Benefit: Cleaner, more flexible design.

---

11) Abstract Base Classes (interfaces / contract)

Core: Force subclasses to implement methods.

Example:

```
from abc import ABC, abstractmethod

class Worker(ABC):
    @abstractmethod
    def work(self): pass

class Dev(Worker):
    def work(self): print("coding")
```

Benefit: Enforce required interface; useful in large apps.

---

12) Error handling & invariants

Core: Methods should check inputs, raise exceptions on bad data.

Example:

```
class Rectangle:
    def __init__(self, w, h):
        if w<=0 or h<=0: raise ValueError("invalid dimensions")
        self.w, self.h = w, h
```

Benefit: Objects always valid → easier debugging.

---

13) Testing & `__repr__` for debugging

Core: Good `__repr__` helps tests and REPL debugging.

```
class Person:
    def __init__(self, n, a): self.name, self.age = n, a
    def __repr__(self): return f"Person({self.name!r},{self.age!r})"
```

Benefit: Inspect objects easily.

---

14) Method overriding vs overloading

Core: Python supports overriding (subclass provides new behavior). Overloading (same name diff signature) not native — use default args or `*args`/`**kwargs`.

Example override:

```
class A:
    def greet(self): print("hi")
class B(A):
    def greet(self): print("hello")
```

Method overloading pattern:

```
def func(a, b=None):
    if b is None:
        # behave like single-arg
        pass
```

---

15) Practical pitfalls & best practices (short)

- **Don't expose internals:** use properties/getters, avoid public attributes for invariants.
- **Mutable class vars:** be careful — usually want instance vars.
- **Deep inheritance:** can cause tight coupling — favor composition.
- **Name-mangling is not absolute privacy:** `obj._ClassName__attr` still works — just avoid.
- **Keep single responsibility:** class should do one main job.
- **Small, tested methods:** easier to maintain.

Quick memory cheats (always remember)

- **E A I P** → Encapsulation, Abstraction, Inheritance, Polymorphism

- `self` = current object; `__init__` = setup; `__str__`/`__repr__` = readable; `__add__` = +
- Prefer **composition** for "has-a", **inheritance** for "is-a".
- Use **properties** for controlled attribute access.

Short complete example (BankAccount) — integrates many pillars

```
from abc import ABC, abstractmethod

class Account(ABC):
    def __init__(self, owner):
        self._owner = owner
        self._balance = 0

    def get_owner(self): return self._owner

    def get_balance(self): return self._balance

    def deposit(self, amount):
        if amount <= 0: raise ValueError("positive only")
        self._balance += amount

    @abstractmethod
    def withdraw(self, amount): pass

class Savings(Account):
    def __init__(self, owner, interest_rate=0.02):
        super().__init__(owner)
        self.interest_rate = interest_rate

    def withdraw(self, amount):
        if amount > self._balance: raise ValueError("Insufficient")
        self._balance -= amount

    def apply_interest(self):
        self._balance += self._balance * self.interest_rate

# usage
s = Savings("Nitesh")
s.deposit(1000)
s.withdraw(300)
```

```
print(s.get_balance())  # 700
```

This example shows: encapsulation (`_balance`), abstraction (user calls `deposit/withdraw` only), inheritance (`Savings` extends `Account`), abstract base class (`Account` enforces withdraw).

Final short checklist — when to use what (practical)

- Want grouping of data+behavior → **use class**.
- Want to hide complexity → **abstraction + encapsulation**.
- Need code reuse across similar types → **inheritance** (but check relation).
- Need many unrelated types with same behavior → **polymorphism / duck typing**.
- Want safe attribute control → **properties**.
- Want operations like `+` or `len()` → implement **dunder methods**.

---

constructor

🔎 What is "dunder"?

- **dunder = double underscore**.
- It's just a nickname Python developers use.
- Example:
    - `__init__` → "dunder init"
    - `__str__` → "dunder str"
    - `__len__` → "dunder len"

So **dunder methods** are special methods surrounded by **double underscores** before and after.

🔎 Why do we use them?

These methods let you define how your class behaves with **built-in Python operations**.

✅ Common Dunder Methods

`__init__` → constructor (called when you create an object).

```
class A:
  def __init__(self, x):
    self.x = x
```

Perfect 👍 you're asking the right thing: **how exactly these dunder methods run behind the scenes**. Let's go one by one with step-by-step execution.

🔹 1. `__str__` → string representation

```
class A:

  def __init__(self, x):

    self.x = x
```

```
    def __str__(self):
        return f"My object with x={self.x}"

obj = A(10)
print(obj)
```

Step by step:

1. `obj = A(10)` → constructor `__init__` runs, stores `x=10`.
2. `print(obj)` → Python internally calls `obj.__str__()`.
3. That method returns the string `"My object with x=10"`.

Print shows:

My object with x=10

4.

👉 Without `__str__`, it would print something ugly like `<__main__.A object at 0x0000023>`.

🔹 2. `__len__` → length

```
class A:
    def __len__(self):
        return 42

obj = A()
print(len(obj))
```

Step by step:

1. `len(obj)` → Python checks if class `A` has a `__len__` method.
2. Finds it → calls `obj.__len__()`.
3. That returns `42`.
4. So `len(obj)` = `42`.

👉 This is why lists, strings, etc. can be used with `len()`: they implement their own `__len__`.

🔹 3. `__add__` → addition (+)

```
class A:
    def __init__(self, x):
        self.x = x

    def __add__(self, other):
        return self.x + other.x

a1 = A(5)
a2 = A(7)
print(a1 + a2)
```

Step by step:

1. `a1 = A(5)` → stores `x=5`.
2. `a2 = A(7)` → stores `x=7`.

`a1 + a2` → Python translates this to:

a1.__add__(a2)

3.
4. Inside `__add__`, it does `self.x + other.x` = `5 + 7`.

5. Returns `12`.

So output is:

12

6.

📌 In short

- `print(obj)` → calls `obj.__str__()`.
- `len(obj)` → calls `obj.__len__()`.
- `obj1 + obj2` → calls `obj1.__add__(obj2)`.

That's how Python maps **normal syntax** → to **dunder method calls** automatically.

⚡ Why double underscores?

- It separates these **special methods** from normal methods.
- Python interpreter looks for these names specifically.

Example: when you do:

print(my_obj)
Python automatically calls:

my_obj.__str__()

-

📌 In short:

- Called **dunder** because of **“double underscore”**.
- They’re special methods that Python **auto-calls** for built-in behavior.

ONLY ONE __INIT__ CONSTRUCTOR ALLOWED !!!

🔎 Your code

```
class Employee():

    def __init__(self, name, salary):

        print("i am creating constructor")

        self.name = name

        self.salary = salary


    def __init__(self):

        print("yes, hey m coming again")
```

❌ What happens here

- In Python, you **cannot** have two constructors (`__init__`) in the same class like in Java or C++.
- The **second `__init__` overrides the first one**.

So when you run:

e = Employee()

Output will be:

yes, hey m coming again

-

But if you try:

e = Employee("Nitesh", 5000)

You'll get:

TypeError: Employee.__init__() takes 1 positional argument but 3 were given

- because only the second constructor exists.

✅ Correct ways to achieve "multiple constructors"

Use default arguments

```
class Employee:
    def __init__(self, name=None, salary=None):
        if name is not None and salary is not None:
            print("Constructor with name and salary")
            self.name = name
            self.salary = salary
        else:
            print("Default constructor")
            self.name = "Unknown"
            self.salary = 0
```

Usage:

```
e1 = Employee("Nitesh", 5000)   # Constructor with name and salary
```

e2 = Employee() # Default constructor

1.

Use class methods as alternative constructors

```
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

    @classmethod
    def from_string(cls, data):
        name, salary = data.split("-")
        return cls(name, int(salary))
```

Usage:

```
e1 = Employee("Nitesh", 5000)
e2 = Employee.from_string("Harry-6000")
```

2.

📌 **Summary**:

- Only **one** `__init__` is allowed — the last one overrides earlier ones.
- To simulate multiple constructors → use **default arguments** or **class methods**.

__init__ ✅

⚡ Key Point

- ✅ Correct constructor: `__init__` (two underscores **before and after**).
- ❌ Wrong: `init` or `__init` (missing underscores).

📌 Summary

- `__init__` → reserved by Python, auto-called as constructor.
- `__kmi__` (or any random dunder) → just a normal method, not special unless Python defines it in its rules.

1️⃣ First version

```
class Employee():
    def init(self, name, salary):   # not __init__
        ...
```

- This is **just a normal method** named `init`, not a constructor.
- Python will not treat it as a special method.

When you create an object like:

```
e = Employee("Nitesh", 5000)
```

❌ It will throw an error:

```
TypeError: Employee() takes no arguments
```

-
- Because the constructor (`__init__`) is missing.

2️⃣ Second version

```
class Employee():
```

```
def __init(self, name, salary):   # only 2 underscores at start
    ...
```

- Here you used **`__init`** (two underscores at start, none at end).
- Python does not recognize this as the constructor.

So again:

```
e = Employee("Nitesh", 5000)
```

- ❌ Error: no valid constructor.

3️⃣ Correct version ✅

```
class Employee():
    def __init__(self, name, salary):   # two underscores before AND after
        self.name = name
        self.salary = salary
```

- This is the real **dunder method** (`__init__`).

Python automatically calls it when you create an object:

```
e = Employee("Nitesh", 5000)
print(e.name, e.salary)
```

- ✅ Works fine.

⚡ Key Point

- ✅ Correct constructor: `__init__` (two underscores **before and after**).
- ❌ Wrong: `init` or `__init` (missing underscores).

📌 Summary

- `__init__` → reserved by Python, auto-called as constructor.
- `__kmi__` (or any random dunder) → just a normal method, not special unless Python defines it in its rules.

🔎 `__init__` is special

- Python only treats **specific dunder names** as "special methods" (`__init__`, `__str__`, `__len__`, etc.).

If you write:

```
def __init__(self, name):
    ...
```

- Python knows this is the **constructor** and will auto-call it when you do `Employee("Nitesh")`.

🔎 If you write `__kmi__`

```
class Employee:
    def __kmi__(self, name):
        print("Hello", name)

e = Employee("Nitesh")
```

❌ This will throw an error:

TypeError: Employee() takes no arguments

Because Python doesn't know `__kmi__` is supposed to be the constructor.
It only understands `__init__` as the constructor.

✅ But you *can* define `__kmi__`

You can make any method name you want, even with double underscores:

```
class Employee:

    def __init__(self, name):

        self.name = name


    def __kmi__(self):

        print("This is my custom dunder method for", self.name)


e = Employee("Nitesh")

e.__kmi__()
```

Output:

```
This is my custom dunder method for Nitesh
```

So `__kmi__` works, but it's just a **normal method with a funky name**.
Python will **not call it automatically** — you must call it yourself.

📌 Summary

- `__init__` → reserved by Python, auto-called as constructor.
- `__kmi__` (or any random dunder) → just a normal method, not special unless Python defines it in its rules.

__init__(* and **)

Excellent question 🔥 — and this shows you're *thinking like a Python developer,* not just memorizing.

Let's break this very clearly 👇 — difference between:

def __init__(self, *args)
def __init__(self, **kwargs)

and why we use * (single star) or ** (double star).

🧠1️⃣ What `*args` and `**kwargs` Actually Mean

Syntax	Meaning	Data Type Inside Function	Used For
`*args`	Variable **positional arguments**	tuple	When you don't know how many positional arguments will come
`**kwargs`	Variable **keyword arguments**	dict	When you don't know how many *named* arguments will come

---

🔹 Example 1 — `*args` (Single Star)

```
class Demo:
    def __init__(self, *args):
        print(args)

d = Demo(10, 20, 30)
```

✅ Output:

(10, 20, 30)

Explanation:

- Everything you pass **without names** goes into a **tuple** called `args`.
- So inside function:
  `args = (10, 20, 30)`

Usage:

- When order matters and you don’t want to name parameters.

You can loop over them easily:

```
for item in args:
    print(item)
```

-

🔹 Example 2 — `**kwargs` (Double Star)

```
class Demo:
    def __init__(self, **kwargs):
        print(kwargs)

d = Demo(name="Nitesh", age=21, course="Python")
```

✅ Output:

```
{'name': 'Nitesh', 'age': 21, 'course': 'Python'}
```

Explanation:

- Everything you pass as **key=value** goes into a **dictionary**.
- So inside function:
  `kwargs = {'name': 'Nitesh', 'age': 21, 'course': 'Python'}`

💡 Why “two stars” (**)

- `*` → unpacks a sequence like list/tuple
- `**` → unpacks a dictionary (key=value pairs)

When we write:

```
def __init__(self, **kwargs):
```

it means:

> “Accept any number of keyword (named) arguments, and store them in a dictionary called `kwargs`.”

Then we can do:

```
for key, value in kwargs.items():
    setattr(self, key, value)
```

🔄 Comparison Table

Feature	`*args`	`**kwargs`
Collects	Positional args	Keyword args
Stored as	Tuple	Dictionary
Example call	`func(1,2,3)`	`func(x=1, y=2)`
Example inside	`args = (1,2,3)`	`kwargs = {'x':1,'y':2}`
Common Use	When argument count unknown	When keys unknown
Unpacking syntax	`*sequence`	`**dictionary`

🧩 Real Use Together

You can even combine both in one constructor:

```
class Example:
    def __init__(self, *args, **kwargs):
        print("Args:", args)
        print("Kwargs:", kwargs)

e = Example(1, 2, 3, name="Nitesh", age=21)
```

✅ Output:

```
Args: (1, 2, 3)
```

Kwargs: {'name': 'Nitesh', 'age': 21}

🚀 Real Company Use Case Example

When a class has to accept **any configuration or settings**:

```
class Config:
    def __init__(self, **kwargs):
        for k, v in kwargs.items():
            setattr(self, k, v)

api_config = Config(api_key="1234", timeout=30, retries=5)
print(api_config.__dict__)
```

✅ Output:

{'api_key': '1234', 'timeout': 30, 'retries': 5}

✅ In Short

Symbol	Means	Example	Type
`*`	Collect/Unpack positional args	`(1, 2, 3)`	tuple
`**`	Collect/Unpack keyword args	`{'x':1, 'y':2}`	dict

Encapsulation

🔹 Encapsulation in Python Recap

- **Encapsulation** = data + methods ek jagah (inside class).
- Access ko control karne ke liye Python me modifiers ka system hai:
    - **Public** → `self.name`
    - **Protected** → `self._name`
    - **Private** → `self.__name`

🔹 Context 1: Public (default)

Sabse simple: **sabko access hai**, koi restriction nahi.

Example:

```
class Student:
    def __init__(self, name):
        self.name = name   # public

s = Student("Nitesh")
print(s.name)  # allowed
s.name = "Amit"  # allowed
print(s.name)
```

👉 **Use when:** attribute harmless hai, aur user freely change kar sakta hai.
👉 **Downside:** data galat state me jaa sakta hai.

🔹 Context 2: Protected (single underscore `_`)

Convention: **class ke andar ya subclass ke andar use karna chahiye**.
Outside se access ho sakta hai, but Python community मानती hai → "isko directly mat chhedo".

Example:

```
class Student:
    def __init__(self, name):
        self._name = name  # protected

    def show(self):
        print("Student:", self._name)

class CollegeStudent(Student):
    def details(self):
        print("Accessing protected:", self._name)

c = CollegeStudent("Nitesh")
c.show()
c.details()

print(c._name)  # allowed but not recommended
```

👉 **Use when:** attribute internal logic ke liye hai, subclasses ko use karna hai, par external user ko discourage karna hai.
👉 **Downside:** ye sirf convention hai, actual restriction nahi.

Mine -

Like - this still we can modify or access **protected object**

```
class student():

    def __init__(self,name):

        self._name = name


    def show(self):

        print("Student :",self._name)


class college(student):

    def details(self):

        print("Accessing protected:", self._name)
```

```
c = college("Nitesh")

print(c._name)

c.show()

c.details()

c._name = "Ram"

c.show()

c.details()
```

🔹 Context 3: Private (double underscore `__`)

Yaha Python **name mangling** apply karta hai → attribute ka naam internally change ho jata hai `_ClassName__attr`.

Example:

```
class Student:
    def __init__(self, name):
        self.__name = name   # private

    def get_name(self):
        return self.__name

s = Student("Nitesh")
print(s.get_name())    # allowed
# print(s.__name)      # ❌ AttributeError

print(s._Student__name)  # 😈 possible, but not recommended
```

```
class student():
    def __init__(self,name):
        self.__name = name

    def get_name(self):
        return f"Student Name : {self.__name}"
```

```
s = student("Nitesh")
#s._student__name = "ram" -- by doing this we can also change private...
print(s.get_name())
```

👉 **Use when:** tum chahte ho koi bhi bahar se galti se bhi variable access/modify na kare.
👉 **Downside:** not 100% private (Python philosophy: “we are all consenting adults here”).

🔹 Context 4: With Methods (safe encapsulation)

Best practice: private/protected + getter/setter methods ya `@property`.

Example with property:

```
class BankAccount:
    def __init__(self, bal):
        self.__balance = bal   # private

    @property
    def balance(self):   # getter
        return self.__balance

    @balance.setter
    def balance(self, amt):   # setter
        if amt < 0:
            raise ValueError("Balance cannot be negative")
        self.__balance = amt

acct = BankAccount(1000)
print(acct.balance)   # get
acct.balance = 2000   # set
# acct.balance = -500  # ❌ error
```

👉 Yaha encapsulation ka asli maza: data protected hai, aur controlled access milta hai.

🔹 Real-life Understanding (Contexts)

- **Public (`self.name`)** → Jab tumhara data harmless hai, jaise "student name", "car color".
- **Protected (`self._name`)** → Jab data internal hai, par subclasses ko chahiye, jaise "engine settings", "temporary cache".
- **Private (`self.__name`)** → Jab tumhe data ko strictly safe rakhna hai, jaise "bank balance", "password".

🔑 Core Idea to Never Forget

Encapsulation ka matlab hai:

- Data ko ek jagah band karo class ke andar.
- Access modifiers (public/protected/private) se decide karo **kisko access dena hai aur kaise**.
- Python trust karta hai developer pe (isliye `_` sirf convention hai), par `__` se accidental misuse avoid hota hai.

Inheritance

Inheritance overall...

🌳 What is Inheritance?

Inheritance allows a **child class** to reuse the properties and methods of a **parent class**.
It makes code DRY (Don't Repeat Yourself) and easy to extend.

⚙️ Syntax

```
class Parent:
    # parent methods & variables

class Child(Parent):
    # child methods & variables
```

🧩 5 Types of Inheritance in Python

Type	Example
1️⃣ Single Inheritance	One parent → one child
2️⃣ Multiple Inheritance	One child → many parents
3️⃣ Multilevel Inheritance	Grandparent → Parent → Child
4️⃣ Hierarchical Inheritance	One parent → many children
5️⃣ Hybrid Inheritance	Combination of any two/more above

💥 Plus — `super()`

The `super()` function is used to **call the parent class's constructor or methods** inside the child class.

🧩 Bonus — `super()` Rules Recap

MRO (Method Resolution Order) AND SUPER() USES

Use Case	Description
`super().__init__()`	Calls parent constructor
`super().method()`	Calls parent's method
Works with MRO	In multiple inheritance, it follows MRO order
Best practice	Always prefer `super()` over hardcoding parent class name

MRO (Method Resolution Order) AND SUPER() USES

🧠 Soch lo aapka example:

Suppose aap likhte ho:

```
class Father1:
    def skills(self):
        print("Business", end=" ")

class Father2:
    def skills(self):
        print("Driving", end=" ")

class Mother1:
    def skills(self):
        print("Cooking", end=" ")

class Child(Father1, Father2, Mother1):
    def skills(self):
        super().skills()
        print("+ Coding")
```

🔍 Step 1: Python banata hai MRO

Inheritance order likha hai:

Child(Father1, Father2, Mother1)

So MRO (method resolution order) hota hai:

[Child, Father1, Father2, Mother1, object]

🔍 Step 2: Jab aap likhte ho `super().skills()`

Toh Python dekhta hai → “Child ke baad MRO me kaun hai?”
Answer: **Father1**

So it calls → `Father1.skills(self)`

MRO (Method Resolution Order) AND SUPER() USES

🧩 Step 3: Agar `Father1` ke andar bhi `super().skills()` likha ho,

tab hi Python automatically next class (Father2) pe jaayega.

Otherwise, **chain yahin ruk jaayegi.**

🧠 Example 1 — Without super() in all parents:

```
class Father1:
    def skills(self):
        print("Business", end=" ")

class Father2:
    def skills(self):
        print("Driving", end=" ")

class Mother1:
    def skills(self):
        print("Cooking", end=" ")

class Child(Father1, Father2, Mother1):
    def skills(self):
        super().skills()
        print("+ Coding")

c = Child()
c.skills()
```

👉 Output:

Business + Coding

Explanation:
Only `Father1` runs, because `Father1` didn't call `super()` to continue the chain.

MRO (Method Resolution Order) AND SUPER() USES

🧠 Example 2 — With `super()` chain maintained properly:

```
class Father1:
    def skills(self):
        print("Business", end=" ")
        super().skills()

class Father2:
    def skills(self):
        print("Driving", end=" ")
        super().skills()

class Mother1:
    def skills(self):
        print("Cooking", end=" ")

class Child(Father1, Father2, Mother1):
    def skills(self):
        super().skills()
        print("+ Coding")

c = Child()
c.skills()
```

👉 Output:

Business Driving Cooking + Coding

Explanation:
Now each parent used `super().skills()`,
so Python followed MRO fully:
`Child → Father1 → Father2 → Mother1 → object`

💡 So the rule:

Situation	What happens
Only child calls `super()`	Only first parent in MRO runs

MRO (Method Resolution Order) AND SUPER() USES

Every parent also calls `super()`	Full MRO chain executes automatically
You skip `super()` in between	Chain breaks there
You call `Father2.skills(self)` manually	You break MRO manually and control goes directly to that class

---

✅ Summary (Hinglish version):

- MRO = order jisme Python methods dhundhta hai.
- `super()` = next class in MRO ko call karta hai.
- Agar saare parents `super()` use karein, toh Python automatically sabko call karega.

Agar koi parent `super()` use nahi karta, toh aapko manually likhna padega jaise:

```
 Father2.skills(self)
Mother1.skills(self)
```

- But that's not recommended, kyunki ye MRO chain tod deta hai.

Would you like me to show a **diagram** (flow arrows) of how MRO flows with and without `super()`?
(It helps you visualize exactly how Python travels through classes.)

Polymorphism

Polymorphism in Python — Easy Notes (Hinglish Style)

1️⃣ Definition:

Polymorphism ka matlab hai:

> “Ek hi cheez (function/operator) alag-alag form me behave kare depending on context.”

Python me polymorphism mainly 4 types me hota hai:

2️⃣ Types of Polymorphism

A) Built-in Polymorphism

- **Meaning:** Python ke operators aur functions **different data types ke saath behave differently** karte hain.
- **Example:**

```
print(5 + 10)       # number add
print("Hi " + "Bro") # string concat
print([1]+[2,3])    # list merge
print(len("Python")) # 6
print(len([1,2,3]))  # 3
```

> 💡 Tip: Ye simplest form, Python me already built-in hota hai.

B) Method Overriding (Runtime Polymorphism)

- **Meaning:** Parent class me jo method hai, child class me **same method name redefine kar do**.
- **Example:**

```
class Bank:
    def roi(self):
        return 0

class SBI(Bank):
    def roi(self):
        return 5.5

print(SBI().roi())  # 5.5
```

> 💡 Tip: Runtime pe decide hota hai kaunsa method call hoga.

C) Function Overloading (Simulated in Python)

- **Meaning:** Python me **true overloading nahi hai**, but `*args` ya default parameters se simulate kar sakte ho.
- **Example:**

```
class Math:
    def add(self, a=None, b=None, c=None):
        if a!=None and b!=None and c!=None: return a+b+c
        elif a!=None and b!=None: return a+b
        else: return "Invalid"

print(Math().add(2,3))     # 5
print(Math().add(1,2,3))   # 6
```

> 💡 Tip: Number of arguments ya type ke hisaab se function ka behaviour change ho jata hai.

D) Duck Typing (Dynamic Polymorphism)

- **Meaning:** Python me type checking nahi hoti, bas object ke **methods/attributes ka existence matter karta hai**.
- **Phrase:** “If it walks like a duck & quacks like a duck, it’s a duck 🦆”

```
class Pycharm:
    def execute(self): print("Run code")
```

```
class VSCode:
    def execute(self): print("Run + lint + auto save")

def programmer_ide(ide):
    ide.execute()

programmer_ide(Pycharm())  # Run code
programmer_ide(VSCode())   # Run + lint + auto save
```

> 💡 Tip: Duck typing → type se nahi, behaviour se kaam hota hai.

E) Polymorphism with `super()`

- **Meaning:** Child class me agar parent ka method ya constructor bhi call karna ho, to `super()` use karte hain.
- **Example:**

```
class Vehicle:
    def show(self): print("Vehicle")

class Car(Vehicle):
    def show(self):
        super().show()  # Parent method
        print("Car")

Car().show()
```

> ✅ Output:

```
Vehicle
Car
```

> 💡 Tip: Ye **inheritance + overriding** ke combination me bohot useful hai.

3️⃣ Quick Differentiation Table

Type	Concept	Example / Key Point

Built-in	Operator/function behaves differently for different types	`+`, `len()`
Overriding	Child redefines parent method	`Bank -> SBI roi()`
Overloading	Same method name, different params	`add(a,b)` vs `add(a,b,c)`
Duck Typing	Type flexible, method existence matter	`ide.execute()`
Super + Polymorphism	Call parent method inside child	`super().show()`

4️⃣ Memory Tips / Tricks

1. **Built-in** → operators/functions ka magic
2. **Overriding** → parent method replace child me
3. **Overloading** → same name, different arguments
4. **Duck Typing** → behaviour > type
5. **Super** → parent ko bhi call karna

💡 **Trick to remember:**

BODS → **B**uilt-in, **O**verriding, **D**uck, **S**uper

- Overloading can be remembered as “args count change”

Abstractions

Abstraction ka matlab:

Ek **blueprint** banana — jisse subclasses follow karein.
Yani: *“Rule set kar do, implementation baad mein batao.”*

🔹 Code samajh lo:

```
from abc import ABC, abstractmethod

class Bank(ABC):
    def __init__(self, name):
        self.name = name

    @abstractmethod
    def roi(self):
        pass
```

Ye `Bank` class bolti hai:

> “Jo bhi mera child hoga (SBI, HDFC...), usko `roi()` method banana hi padega.”

Agar nahi banaya → ❌ error.

🔹 Fayda kya hai?

1. **Rule enforce hota hai** – har bank ke paas `roi()` zaroor hoga.

Common interface milta hai – sabko ek jaisa treat kar sakte ho:

```
for bank in [SBI("SBI"), HDFC("HDFC")]:
    print(bank.roi())
```

2.
3. **Code maintain karna easy** – naye banks add karo bina purana code badle.

🔹 Ek line mein:

👉 **Abstraction = Rule + Structure + Future safety.**
Yani sab banks ek pattern follow karein — koi apni marzi se method na bhool jaye 😄

#but abstraction means to hide krna important cheej ko ye hota h na

Bahut acha sawal hai 👏 — aur ye confusion **sabke saath hota hai!**
Chalo short aur simple mein samjhte hain 👇

🔹 "Abstraction" ka real matlab

Haan, **abstraction ka basic idea hota hai — "cheezon ko hide karna"**
par iska matlab hota hai:

> *User ko sirf important cheez dikhana, aur complex cheez chhupana.*

🔹 Example se samjho:

Jab tum **ATM machine** use karte ho:

- Tum sirf button dabate ho aur paisa nikalta hai 💸
- Tumhe nahi pata andar machine kaun sa circuit use kar rahi hai, ya data server pe kaise ja raha hai.

➡️ Yehi **abstraction** hai — *andar ki complex logic chhupai gayi hai.*

🔹 Programming mein kaise hota hai:

`abstract class` ek **blueprint** hoti hai —
jo kehti hai "*main rule bataungi, implementation baad mein hide rahega*".

```
class Bank(ABC):
    @abstractmethod
    def roi(self):
        pass
```

Yahan `roi()` ka **logic hide** hai.
Bas itna dikhaya gaya hai ki "har bank me roi hoga."

Phir subclasses (SBI, HDFC) apna logic likhti hain.
User ko bas ye maloom hai — `bank.roi()` call karna hai.

🔹 Short summary:

Concept	Matlab
Abstraction	Complex cheez hide karke sirf important part dikhana
Abstract Class	Rule define karti hai, implementation hide karti hai
Encapsulation	Data ko ek jagah band karke safe rakhna (jaise private variables)

👉 So tum bilkul sahi soch rahe ho —
Abstraction = Hide karna important logic ko
lekin **Abstract Class = Us abstraction ko implement karne ka ek tareeka** hota hai.

Feature	Abstract Class	Notes
Object creation	❌ Not allowed	Only child objects can be created
Abstract method	`@abstractmethod`	Must be implemented in child
Normal method	Allowed	Can be inherited as-is
Purpose	**Hide implementation, show only essential**	Blueprint/interface style
Use case	Payment systems, shapes, bank systems	Ensures uniform method names across child classes

🧩 Q5 — Real-World Mini Project: Payment System

Question:
Abstract class `Payment` with abstract method `pay(amount)`

- Child classes: `UPI`, `Card`, `NetBanking` → implement `pay()` differently

- Demonstrate **polymorphic behaviour** using list of objects.

✅ Solution:

```
from abc import ABC, abstractmethod

class Payment(ABC):
    @abstractmethod
    def pay(self, amount):
        pass

class UPI(Payment):
    def pay(self, amount):
        print(f"Paid ₹{amount} via UPI")

class Card(Payment):
    def pay(self, amount):
        print(f"Paid ₹{amount} via Card")

class NetBanking(Payment):
    def pay(self, amount):
        print(f"Paid ₹{amount} via NetBanking")

# Testing
payments = [UPI(), Card(), NetBanking()]
for p in payments:
    p.pay(1000)
```

✅ Output:

```
Paid ₹1000 via UPI
Paid ₹1000 via Card
Paid ₹1000 via NetBanking
```

🧠 **Concept:**

- Abstract class ensures all child classes implement `pay()`
- List of abstract type → **polymorphic calls**

short and proper

🧠 Abstraction in Python — Complete Notes (Hinglish + English)

🔹 1️⃣What is Abstraction?

👉 **Definition:**
Abstraction ka matlab hota hai — “User ko sirf important cheez dikhana aur internal working ko hide kar dena.”

Example:
Tu jab ATM me paisa nikalta hai —

- Tu sirf amount daalta hai (interface).
- System andar complex logic chalata hai (implementation hidden).
  ➡️ Yehi abstraction hai.

🔹 2️⃣Why Abstraction in Python?

✅ Hide complex logic
✅ Force child classes to implement specific methods
✅ Maintain uniform structure across multiple classes (blueprint)
✅ Prevent direct object creation from base class

🔹 3️⃣How to Achieve Abstraction?

In Python, abstraction achieve hota hai using the **`abc` module** (Abstract Base Class).

```
from abc import ABC, abstractmethod
```

- `ABC` → Abstract Base Class
- `@abstractmethod` → decorator used to declare abstract methods

🔹 4️⃣ Rules of Abstraction

Rule	Explanation
1️⃣	Abstract class cannot be instantiated (object can't be created)
2️⃣	Abstract method must be implemented in child class
3️⃣	Abstract class can also have normal (concrete) methods
4️⃣	If any abstract method left unimplemented → Error
5️⃣	Used for designing **blueprints or interfaces**

🔹 5️⃣ Basic Example

```
from abc import ABC, abstractmethod

class Vehicle(ABC):
    @abstractmethod
    def wheels(self):
        pass

class Bike(Vehicle):
    def wheels(self):
        print("Bike has 2 wheels")

class Car(Vehicle):
    def wheels(self):
        print("Car has 4 wheels")

b = Bike()
c = Car()
b.wheels()
c.wheels()
```

🧩 **Explanation:**

- `Vehicle` is abstract → object cannot be made
- `wheels()` method **must** be implemented by all subclasses

🔹 6️⃣ Abstract Class with Constructor

```
from abc import ABC, abstractmethod

class Bank(ABC):
    def __init__(self, name):
        self.name = name

    @abstractmethod
    def roi(self):
        pass

class SBI(Bank):
    def roi(self):
        return 5.5

class HDFC(Bank):
    def roi(self):
        return 6.2

for bank in [SBI("SBI"), HDFC("HDFC")]:
    print(f"{bank.name} ROI: {bank.roi()}%")
```

📘 **Concept:**

- Constructor normal tarah se kaam karta hai
- Only abstract method ko implement karna mandatory hai

🔹 7️⃣ Abstract + Normal Methods Together

```
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

    def info(self):
        print("This is a shape class")

class Rectangle(Shape):
    def __init__(self, w, h):
        self.w = w
        self.h = h
```

```
    def area(self):
        return self.w * self.h

r = Rectangle(5, 10)
r.info()
print("Area:", r.area())
```

🧠 **Concept:**
Abstract class me dono types ke methods ho sakte hain → abstract + normal.

🔹 8️⃣ Real-World Example — Payment System

```
from abc import ABC, abstractmethod

class Payment(ABC):
    @abstractmethod
    def pay(self, amount):
        pass

class UPI(Payment):
    def pay(self, amount):
        print(f"Paid ₹{amount} via UPI")

class Card(Payment):
    def pay(self, amount):
        print(f"Paid ₹{amount} via Card")

class NetBanking(Payment):
    def pay(self, amount):
        print(f"Paid ₹{amount} via NetBanking")

payments = [UPI(), Card(), NetBanking()]
for p in payments:
    p.pay(1000)
```

🧩 **Concept:**

- `Payment` defines interface (blueprint).
- Every payment method **must** define `pay()`.
- This also shows **Polymorphism + Abstraction** working together.

🔹 9️⃣ Difference Between Encapsulation, Abstraction & Polymorphism

Concept	Meaning	Example
Encapsulation	Data hiding using private/protected variables	`self.__balance`
Abstraction	Hiding implementation details	`@abstractmethod`
Polymorphism	Same method, different behaviour	`pay()` in different classes

🧠 **Shortcut to remember:**

Encapsulation = Protect Data
Abstraction = Hide Complexity
Polymorphism = One name, many forms

🔹 🔟 Real-Life Analogy (Easy to Remember)

Real Example	Python Equivalent
ATM Machine	Abstract Class
"Withdraw Cash" Button	Abstract Method
SBI / HDFC Machines	Child Classes implementing method differently

---

💡 In Short (Memory Tip):

“Abstract class = design plan”
“Child class = builder who must follow that plan”
“@abstractmethod = mandatory rule”
“abc = rule book of abstraction”

Would you like me to make a **PDF note version** of this (formatted with boxes and short tables)
so you can **upload on GitHub or read offline**?

Dunder method

🧠 Topic 9 — Dunder / Magic Methods + Operator Overloading in Python

---

🔹 What are Dunder (Magic) Methods?

“Dunder” = Double UNDerscore
They are **special built-in methods** jo start aur end hote hain `__` se.
Example:
`__init__`, `__str__`, `__add__`, `__len__`, `__eq__`, etc.

🔸 Ye methods automatically call hote hain by Python —
tumhe manually call karne ki zarurat nahi hoti.

🔹 Why we use Dunder Methods?

✅ To **customize object behavior**
✅ To **make your own class act like built-in types** (list, int, etc.)
✅ To enable **operator overloading** (`+`, `-`, `*`, `<`, etc.)

🔹 Commonly Used Dunder Methods

Method	Used For	Example
`__init__()`	Constructor	Object initialization
`__str__()`	String representation	`print(obj)`
`__repr__()`	Debug representation	For developers
`__add__()`	`+` operator	`obj1 + obj2`
`__sub__()`	`-` operator	`obj1 - obj2`
`__mul__()`	`*` operator	`obj1 * obj2`
`__eq__()`	`==` operator	Compare objects

`__lt__()`	`<` operator	Less than
`__len__()`	`len(obj)`	Return length
`__call__()`	When object called as function	`obj()`

🔹 Summary Table — Dunder / Magic Methods

Category	Method	Purpose
Constructor	`__init__`	Initialize attributes
Representation	`__str__`, `__repr__`	Object print formatting
Arithmetic	`__add__`, `__sub__`, `__mul__`, `__truediv__`	Operator overloading
Comparison	`__eq__`, `__lt__`, `__gt__`	Compare objects
Utility	`__len__`, `__call__`, `__contains__`	Custom built-in behavior

💡 Hinglish Tips to Remember:

- **Dunder** = “Double underscore wale methods jo Python khud call karta hai.”
- **Operator overloading** = “Apne class objects ko + - * / < > jaise operators samjha dena.”
- **Always return a new object** in overloading methods for reusability.
- Ye concept interview me **favorite** hota hai, specially `__add__`, `__eq__`, `__str__`.

getter_setter

Eg - 1

```
class Account:

    def __init__(self,balance=0):
        self.__balance = balance

    @property
    def balance(self):
        print("Getting Balance...")
        return self.__balance

    @balance.setter
    def balance(self,amt):
        if amt>=0:
            self.__balance += amt

        else:
            print("❌ Negative balance not allowed!")


acc = Account()
acc.balancekldkjfadf = 5000
print("Your balance : ",acc.balance)

acc.balance = 10000
print("Your balance : ",acc.balance)
```

Eg = 2

```
class Employee:
    def __init__(self,name,dept,salary):
        self.name = name
        self._dept = dept
        self.__salary = salary

    @property
    def get_salary(self):
        return self.__salary
```

```
    @get_salary.setter
    def set_salary(self,new_salary):
        self.__salary = new_salary

    def show(self):
        print(f"Name: {self.name}, Dept: {self._dept}, Salary:
{self.__salary}")


e = Employee("Nitesh","HR",120000)
e.show()
e.name = "Rajesh"
e._dept = "It"
e.set_salary = "18220"
e.show()
```

Output -

Name: Nitesh, Dept: HR, Salary: 120000
Name: Rajesh, Dept: It, Salary: 18220

Super().show()

```
class A:
    def show(self):
        print("It's A")
        #super().show()

class B(A):
    def show(self):
        print("It's B")
        super().show()

class C(A):
    def show(self):
        print("It's C")
```

```
        super().show()

class D(B,C):
    def show(self):
        print("It's D")
        super().show()

d = D()
d.show()
```

Output -

```
It's D
It's B
It's C
It's A
```

last extra

OOPS Final Notes (Part 4/4)

🔟 Composition & Aggregation
1️⃣1️⃣ Access Modifiers
1️⃣2️⃣ MRO (Method Resolution Order)
1️⃣3️⃣ Object Lifecycle

🔟 Composition & Aggregation

🔹 Basic Concept

👉 Jab ek class doosri class ke **object ko apne andar use karti hai**, use **"HAS-A Relationship"** kehte hain.
Ye 2 type ke hote hain:

Type	Meaning	Object Lifetime
Composition	Strong relationship — child cannot exist without parent	Dependent
Aggregation	Weak relationship — child can exist independently	Independent

---

🧩 Composition Example

```
class Engine:
    def start(self):
        print("Engine started...")

class Car:
    def __init__(self):
        self.engine = Engine()   # Car has-a Engine

    def drive(self):
        self.engine.start()
        print("Car is moving 🚗")
```

✅ **Hinglish:**
Car ke andar Engine bana hai → agar Car destroy ho gayi, Engine bhi gaya.
👉 Ye hai **Composition (Strong Relationship)**

🧩 Aggregation Example

```
class Engine:
    def start(self):
        print("Engine started...")

class Car:
    def __init__(self, engine):
        self.engine = engine     # Engine passed from outside

    def drive(self):
        self.engine.start()
        print("Car is moving 🚗")

e = Engine()
c = Car(e)
c.drive()
```

✅ **Hinglish:**
Engine alag bana hai aur Car use kar rahi hai → agar Car destroy ho gayi, Engine fir bhi exist karega.
👉 Ye hai **Aggregation (Weak Relationship)**

🧠 Shortcut Trick:

- “Agar object andar banta hai → Composition.”
- “Agar object bahar se aata hai → Aggregation.”

1️⃣1️⃣ Access Modifiers

🔹 Basic Concept

Python mein 3 level ke access modifiers hote hain (though soft rules — not strict like Java).

Type	Symbol	Access	Example
Public	No underscore	Accessible everywhere	`self.name`
Protected	`_single_underscore`	Accessible within class & subclasses	`self._age`

Private	`__double_underscore`	Name mangled (only inside class)	`self.__salary`

🧩 Example

```
class Employee:
    def __init__(self):
        self.name = "Ramesh"       # public
        self._dept = "HR"          # protected
        self.__salary = 50000      # private

e = Employee()
print(e.name)          # ✅ Accessible
print(e._dept)         # ⚠️ Accessible but not recommended
# print(e.__salary)    # ❌ Error

print(e._Employee__salary)  # ✅ Name mangling access
```

✅ Hinglish:

- `Public`: sabko dikh raha hai
- `Protected`: "family members only"
- `Private`: "sirf class ke andar"

💡 Python “trust” based language hai — matlab rules hai, but todne pe koi police nahi 😅

1️⃣2️⃣ MRO (Method Resolution Order)

🔹 Basic Concept

Multiple inheritance mein jab same method alag-alag parent classes mein ho,
Python ko decide karna hota hai **kis class ka method pehle chalega** —
Ye decide karta hai **MRO** (Method Resolution Order).

🧩 Example

```
class A:
    def show(self): print("A")
```

```
class B(A):
    def show(self): print("B")

class C(A):
    def show(self): print("C")

class D(B, C):
    pass

obj = D()
obj.show()
print(D.mro())
```

✅ Output:

```
B
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>,
<class 'object'>]
```

✅ **Hinglish Explanation:**

- Python MRO = D → B → C → A → object
- Pehla milta method run ho jaata hai
- `super()` bhi isi MRO order ko follow karta hai

💡 Check MRO using → `ClassName.mro()` or `ClassName.__mro__`

🧠 Shortcut Trick:

> "Python left se right, top se bottom check karta hai."
> "super() = next class in MRO chain."

1️⃣3️⃣ Object Lifecycle

🔹 Concept

Every object in Python goes through 3 stages:

Stage	Method	Description
Creation	`__init__()`	Called automatically when object created
Representation	`__str__()` / `__repr__()`	Defines print output
Destruction	`__del__()`	Called when object is deleted or GC removes it

---

🧩 Example

```
class Person:
    def __init__(self, name):
        self.name = name
        print(f"{self.name} created ✅")

    def __str__(self):
        return f"Person: {self.name}"

    def __del__(self):
        print(f"{self.name} destroyed ❌")

p1 = Person("Nitesh")
print(p1)
del p1
```

✅ Output:

```
Nitesh created ✅
Person: Nitesh
Nitesh destroyed ❌
```

✅ Hinglish Explanation:

- `__init__()` → janam (birth)
- `__str__()` → jeevan (how it looks)
- `__del__()` → maran (death)

💡 Destructor (`__del__`) ka timing **garbage collector** decide karta hai.

🧠 Shortcut Trick:

> “Init → Str → Del = Birth → Life → Death”

💎 Final Recap Table

Concept	Keyword / Symbol	Summary	Real-world Example
Composition	Object inside class	Strong binding	Heart inside Human
Aggregation	Object passed in class	Loose binding	Engine used by Car
Public	`name`	Access everywhere	Public data
Protected	`_name`	Access in subclass	Internal use only
Private	`__name`	Access only inside class	Confidential data
MRO	`Class.mro()`	Defines inheritance order	Multi-parent conflict solver
super()	Keyword	Calls next class method in MRO	Cooperative inheritance
Object Lifecycle	`__init__`, `__del__`	Create → Use → Destroy	Human birth to death

---

💬 Memory Lines (for fun recall 😄)

> 🧩 “Agar object andar banta hai – Composition.
> Bahar se aata hai – Aggregation.”

> 🔐 “Public sabka, Protected ghar ka, Private sirf room ka.”

> 🌀 “MRO line mein chalata hai – left to right, top to bottom.”

> 🧍 “Init = Janam, Str = Jeevan, Del = Maran.”

Would you like me to now make this note into a **beautiful PDF version (with formatting, emojis, color sections & code boxes)** — perfect for printing or adding to GitHub README?

cwh_adv_py_1_class - 4

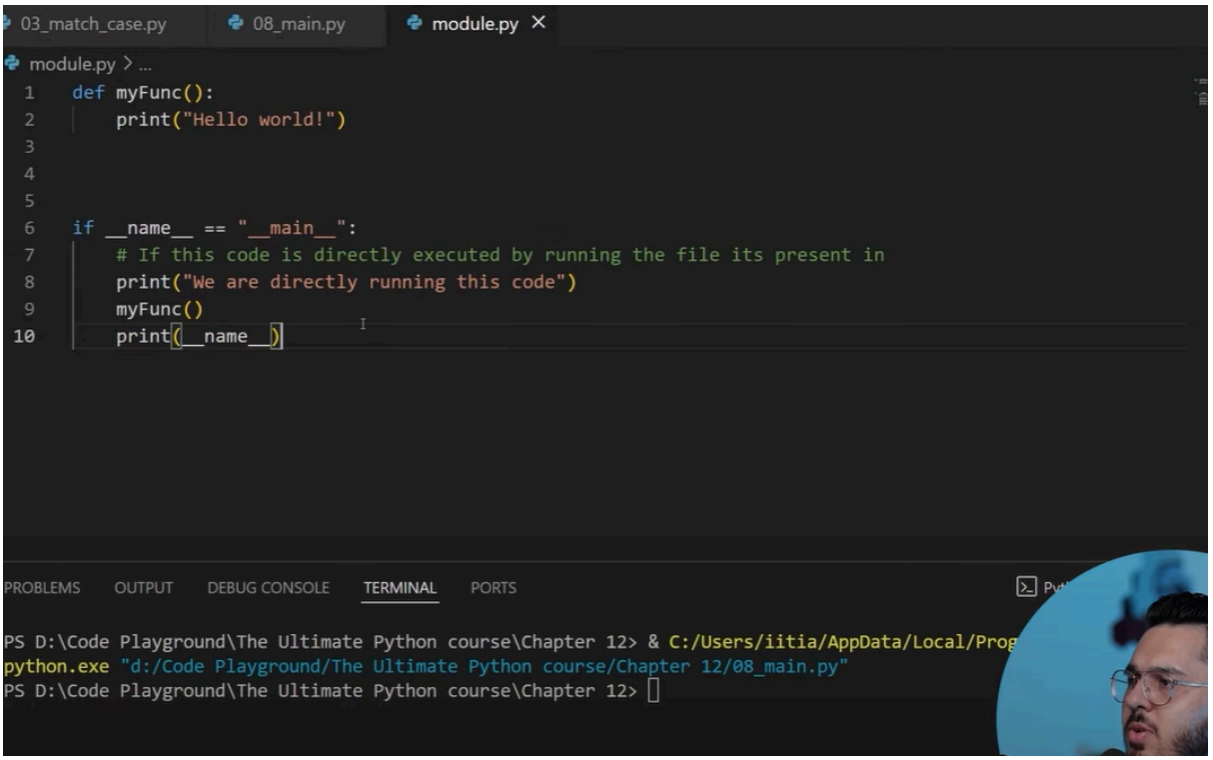
CWH ADVANCED PYTHON 1

1) IF __NAME__== '__MAIN__' IN PYTHON

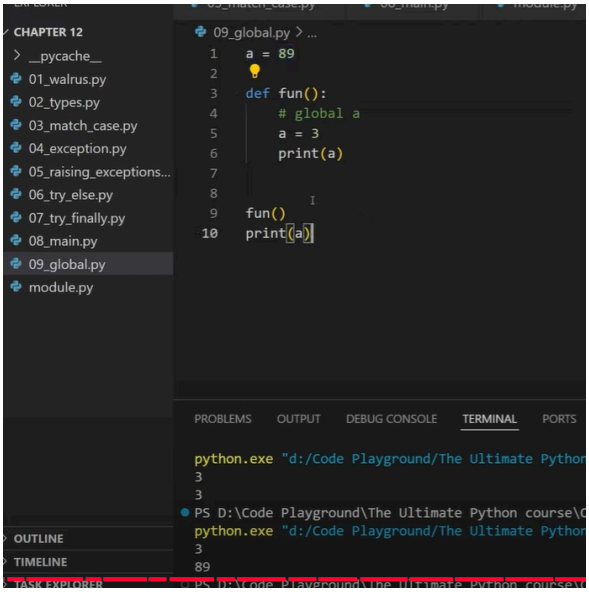
'__name__' evaluates to the name of the module in python from where the program is run. If the module is being run directly from the command line, the '__name__' is set to string "__main__".

Thus, this behaviour is used to check whether the module is run directly or imported to another file.

This is also for like - i don't want to run a specific code if someone importing this file in another file



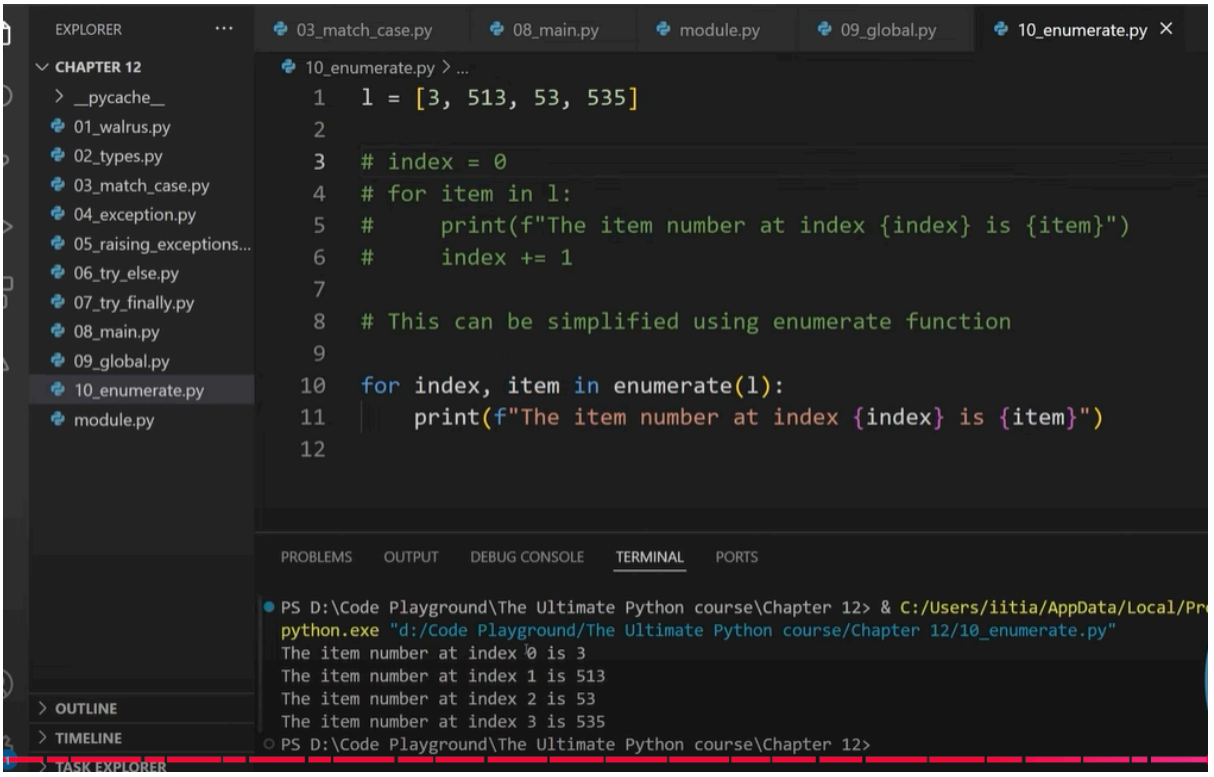
2) THE GLOBAL KEYWORD 'global' keyword is used to modify the variable outside of the current scope.



3) ENUMERATE FUNCTION IN PYTHON

The ‘enumerate’ function adds counter to an iterable and returns it

```
for i,item in list1:
        print(i,item) # Prints the items of list 1 with index
```



4) Jh

5)

cwh_adv_py_2_class - 5

CWH ADVANCED PYTHON 2

1) VIRTUAL ENVIRONMENT

What is this -

This is used for - installing more versions or different different version… we can't use or keep install too many different version in global,
But we also need for different version so we can simple create virtual env..
And keep download all version in that file… and use from there.

How to install virtual environment -

- pip install virtualenv

After it name your virtualenv file name – as you want…

- virtualenv my_new_env
  Here my_new_env === is given file name by me

How to activate virtual environment -

So write - .\env(or name of your virtual env..)\Scripts\activate.ps1

How to deactivate virtual environment -

By just write - deactivate

PIP FREEZE COMMAND

1. pip freeze > requirements .txt #-- for write all module in requirement.txt file
2. pip install –r requirements.txt #-- for install all the module which is in requirement.txt— it will install automatically all module…

For more details - check lib_pip_chatgpt notes in class - 1

2) Map(), Filter(), Reduce()

thisThere are **3 super famous bhai** in Python:

1. `map()` → sab pe **kaam karo.**

   a. **`map()` returns an iterator (lazy sequence).**
   b. **`list(map(...))`, `tuple(map(...))`, or `set(map(...))` convert that iterator into an actual collection in memory.**
   c. **Eg -**

```
a = list(map(square,l))
        print(a)
```

2. `filter()` → sirf **chuno jo pass ho**

```
Eg -
l = [1,2,3,4,5]
even = lambda x: x%2 == 0

print(list(filter(even,l)))
```

3. `reduce()` → sabko **ek mein jodo -**
   for Reduce we have to import ... reduce from Functool.

```
Eg -
from functools import reduce
l = [1,2,3,4,5]
sum_2 = lambda a,b:a+b
print(reduce(sum_2,l))
```

So basically:

map = change all
filter = select few
reduce = combine all

3) adf

‘__nam